TUGAS AKHIR – KI1502

DESAIN DAN ANALISIS ALGORITMA PEMROGRAMAN DINAMIS MODEL TREE PADA PENYELESAIAN PERMASALAHAN SPOJ KLASIK DISJOINT SUBTREES

Fandi Akbar Rahadian
NRP. 5111 100 192

Dosen Pembimbing I
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing II
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

UNDERGRADUATE THESIS – KI141502

DESIGN AND ANALYSIS OF DYNAMIC PROGRAMMING TREE MODEL ALGORITHM FOR SOLVING SPOJ CLASSICAL PROBLEM DISJOINT SUBTREES

Fandi Akbar Rahadian
NRP. 5111 100 192

Advisor I
Arya Yudhi Wijaya, S.Kom., M.Kom.

Advisor II
Rully Soelaiman, S.Kom., M.Kom.

DEPARTMENT OF INFORMATICS
Faculty of Information Technology
Sepuluh Nopember Institute of Technology
Surabaya 2015

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS ALGORITMA PEMROGRAMAN DINAMIS MODEL TREE PADA PENYELESAIAN PERMASALAHAN SPOJ KLASIK DISJOINT SUBTREES

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
Fandi Akbar Rahadian
NRP. 5111100192

Disetujui oleh Pembimbing Tugas Akhir:

1. Arya Yudhi Wijaya, S.Kom., M.Kom. ..... (Pembimbing 1)
   (NIP. *198409042010121002*)
2. Rully Soelaiman, S.Kom., M.Kom. ..... (Pembimbing 2)
   (NIP. *197002131994021001*)

SURABAYA
Juni, 2015

DESAIN DAN ANALISIS ALGORITMA PEMROGRAMAN DINAMIS MODEL TREE PADA PENYELESAIAN PERMASALAHAN SPOJ KLASIK DISJOINT SUBTREES

Nama : **Fandi Akbar Rahadian**
NRP : **5111100192**
Jurusan : **Teknik Informatika**
Fakultas Teknologi Informasi ITS
Dosen Pembimbing I : **Arya Yudhi Wijaya, S.Kom., M.Kom.**
Dosen Pembimbing II : **Rully Soelaiman, S.Kom., M.Kom.**

Abstrak

Diberikan sebuah arbitrary tree dimana setiap vertex pada tree tersebut memiliki bobot tertentu. Tentukan nilai selisih terbesar dari dua himpunan vertex, dimana vertex pada himpunan tersebut tidak dipisahkan oleh suatu vertex yang bukan anggota himpunannya dan kedua himpunan tidak memiliki vertex yang sama. Nilai dari suatu himpunan adalah total jumlah bobot vertex anggotanya.

Pemrograman Dinamis adalah sebuah paradigma untuk mendapatkan nilai optimal dari beberapa kemungkinan jawaban, dimana permasalahan tersebut memiliki submasalah tumpang tindih dan struktur optimal.

Algoritma pemrograman dinamis pada struktur data tree dapat diimplementasikan dengan beberapa cara, menggunakan pemrograman dinamis dalam proses penggabungan nilai submasalah pada children suatu vertex atau dengan mengkonversi arbitrary tree awal menjadi left child right sibling tree agar proses penggabunganan nilai submasalah pada children suatu vertex menjadi lebih sederhana.

Pada penelitian ini didesain dan diimplementasikan solusi untuk permasalahan yang disampaikan pada paragraf pertama dengan pendekatan pemrograman dinamis dalam proses penggabungan nilai submasalah pada children suatu vertex dan pendekatan mengkonversi arbitrary tree menjadi left child right sibling tree.

Solusi yang dikembangkan berjalan dengan kompleksitas waktu $O(NK^2)$*, dimana N adalah jumlah vertex pada arbitrary tree dan K adalah nilai terbesar diantara* K_1 *dan* K_2*.*

Kata kunci : Pemrograman Dinamis, Left Child Right Sibling Tree.

DESIGN AND ANALYSIS OF DYNAMIC PROGRAMMING TREE MODEL ALGORITHM FOR SOLVING SPOJ CLASSICAL PROBLEM DISJOINT SUBTREES

Name	**: Fandi Akbar Rahadian**
NRP	**: 5111100192**
Department	**: Department of Informatics Faculty of Information Technology ITS**
Advisor I	**: Arya Yudhi Wijaya, S.Kom., M.Kom.**
Advisor II	**: Rully Soelaiman, S.Kom,. M.Kom.**

Abstract

Given an arbitrary tree where each vertex in the tree has a certain weight. Determine the value of the largest difference between the two set of vertices, where vertex in the set is not separated by a vertex that is not a member its set and the two sets do not have common vertex. The value of the set is the total number of vertex weights of its members.

Dynamic programming is a paradigm to gain optimal value from several possible answers, where the problem has overlapping subproblems and optimal structure.

Dynamic programming on a tree data structure algorithm can be implemented in several ways, using dynamic programming in the process of merging subproblems value in children of each vertex or to convert the initial tree become left child right sibling tree so that the process of merging subproblems value in children of each vertex become simpler.

In this thesis, will be designed and implemented solution to the problems presented in the first paragraph with a dynamic programming approach in the process of merging subproblems value on children and the approach of converting an arbitrary vertex tree be left child right sibling tree.

The time complexity of the solution is $O(NK^2)$, where N is the number of vertex in the arbitrary tree and K is the largest value of K_1 and K_2.

Keywords : Dynamic Programming, Left Child Right Sibling Tree.

KATA PENGANTAR

Segala puji dan syukur penulis sampaikan ke hadirat Allah SWT yang telah melimpahkan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan penelitian yang berjudul:

DESAIN DAN ANALISIS ALGORITMA PEMROGRAMAN DINAMIS MODEL TREE PADA PENYELESAIAN PERMASALAHAN SPOJ KLASIK DISJOINT SUBTREES

Penulis menyadari bahwa penelitian ini tidak mungkin dapat terselesaikan tanpa bantuan dan dukungan dari banyak pihak, baik secara langsung maupun tidak. Untuk itu, penulis ingin mengucapkan terima kasih dan penghormatan yang sebesar-besarnya kepada :

1. Kedua orang tua penulis yang selalu memberi dukungan, alasan, dan tujuan hingga penulis dapat menyelesaikan penelitian ini.
2. Bapak Arya Yudhi Wijaya, S.Kom., M.Kom., selaku dosen pembimbing penulis. Terima kasih atas masukan-masukan positif yang telah diberikan untuk membuat penelitian ini menjadi lebih baik.
3. Bapak Rully Soelaiman, S.Kom., M.Kom., selaku dosen pembimbing penulis. Terima kasih atas semua ilmu dan pengalaman yang diberikan tentang semua hal, sungguh sangat berharga bagi penulis.
4. Bapak dan Ibu dosen Teknik Informatika ITS. Terima kasih atas segala ilmu yang telah diberikan.
5. Teman-teman kelompok “TA Kace”, Ibet, Kaspul, Kemal, Melfa yang telah saling berbagi suka dan duka selama masa perkuliahan dan pengerjaan penelitian ini.

Penulis telah berusaha menyelesaikan penelitian ini sebaik mungkin, tetapi penulis mohon maaf apabila terdapat kesalahan maupun kelalaian yang penulis lakukan. Penulis mengharapkan kritik dan saran yang dapat membangun sebagai bahan perbaikan selanjutnya.

Surabaya, Juni 2015

Fandi Akbar Rahadian

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR KODE SUMBER

BAB I
PENDAHULUAN

Pada bab ini dijelaskan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi, dan sistematika penulisan penelitian.

1.1 Latar Belakang

Mencari himpunan dengan bobot terbesar pada struktur data *tree* dimana himpunan tersebut diharuskan tidak dipisahkan oleh suatu *vertex* yang bukan himpunannya merupakan sebuah permasalah yang dapat diselesaikan dengan algoritma yang berdasarkan metode pemrograman dinamis.

Topik penelitian ini mengangkat permasalahan pada *Online Judge* SPOJ dengan kode soal KAYKAY yang berjudul *Disjoint Subtrees*. Pada permasalahan ini diberikan sebuah *tree*. Dimana setiap *vertex* pada *tree* memiliki bobot tertentu. Tujuannya adalah mendapatkan nilai selisih terbesar dari dua himpunan *vertex*. Nilai himpunan *vertex* adalah jumlah bobot semua *vertex* yang merupakan anggota himpunan tersebut.

Pada penelitian ini diimplementasikan algoritma pemrograman dinamis dengan dua pendekatan. Pendekatan pertama menggunakan pemrograman dinamis kembali dalam proses penggabungan nilai-nilai submasalah pada *children* setiap *vertex*. Yang kedua, mengkonversi *arbitrary tree* data masukan menjadi *left child right sibling tree*.

1.2 Rumusan Masalah

Perumusan masalah pada penelitian ini adalah sebagai berikut:

1. Bagaimana mendesain dan menganalisis algoritma metode pemrograman dinamis pada Struktur Data Tree

dalam menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees*.

2. Bagaimana melakukan perancangan dan implementasi algoritma metode pemrograman dinamis pada Struktur Data *Tree* dalam menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees.*
3. Bagimana hasil dari kinerja algoritma pemrograman dinamis pada Struktur Data *Tree* dalam menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees*.
4. Bagaimana mendesain generator kasus uji untuk proses uji kinerja metode pemrograman dinamis pada Struktur Data *Tree* dalam menyelesaiakan permasalahan klasik SPOJ *Disjoint Subtrees*.

1.3 Batasan Masalah

Batasan masalah pada penelitian ini adalah sebagai berikut :

1. Implementasi algoritma dilakukan dengan bahasa pemrograman C++.
2. Data masukan dan data keluaran yang digunakan untuk menguji algoritma yang telah dirancang adalah *dataset* SPOJ *Disjoint Subtrees.*
3. Batasan implementasi sesuai dengan permasalah SPOJ Klasik Disjoint Subtrees

1.4 Tujuan

Tujuan dari penelitian ini adalah sebagai berikut :

1. Memahami desain metode pemrograman dinamis pada struktur data *tree* untuk menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees*.
2. Melakukan implementasi metode pemrograman dinamis pada struktur data *tree* dalam berbagai pendekatan untuk menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees*.

3. Mengevaluasi berbagai pendekatan yang berdasarkan metode pemrograman dinamis untuk menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees* dengan melakukan uji coba.
4. Mendesain dan mengimplementasikan generator kasus uji untuk proses uji kinerja metode pemrograman dinamis pada struktur data *tree* dalam berbagai pendekatan untuk menyelesaikan permasalahan klasik SPOJ *Disjoint Subtrees*.

1.5 Metodologi

Berikut metodologi yang digunakan dalam penelitian ini :

1. Penyusunan proposal penelitian
   Pada tahap ini dilakukan penyusunan proposal penelitian yang berisi definisi permasalahan SPOJ Klasik *Disjoint Subtrees* beserta gambaran solusi penyelesaiannya.
2. Studi Literatur
   Pada tahap ini dilakukan studi literatur mengenai permasalahan pemrograman dinamis pada struktur data *tree*. Literatur yang digunakan antara lain buku referensi dan artikel yang didapat dari internet.
3. Desain
   Pada tahap ini dilakukan berbagai desain algoritma dari solusi untuk permasalahan SPOJ Klasik *Disjoint Subtrees*.
4. Implementasi
   Pada tahap ini dilakukan implementasi solusi untuk permasalahan SPOJ Klasik *Disjoint Subtrees* berdasarkan analisis dan desain yang telah dilakukan.
5. Uji coba dan evaluasi
   Pada tahap ini dilakukan uji coba untuk menguji kebenaran dan performa dari implemantasi algoritma yang telah dilakukan.
6. Penyusunan buku penelitian

Pada tahap ini dilakukan penyusunan buku penelitian yang berisi dokumentasi mengenai solusi untuk permasalahan SPOJ Klasik *Disjoint Subtrees*.

1.6 Sistematika Penulisan

Berikut sistematika penulisan buku penelitian ini :

1. **BAB I : PENDAHULUAN**
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, metodologi dan sistematika penulisan penelitian.
2. **BAB II : TINJAUAN PUSTAKA**
   Bab ini berisi dasar teori mengenai permasalahan dan algoritma yang digunakan dalam penelitian.
3. **BAB III : DESAIN**
   Bab ini berisi desain algoritma serta struktur data yang digunakan dalam penelitian.
4. **BAB IV : IMPLEMENTASI**
   Bab ini berisi implementasi berdasarkan desain algoritma serta struktur data yang telah dilakukan.
5. **BAB V : UJI COBA DAN EVALUASI**
   Bab ini berisi uji coba dan evaluasi dari implementasi yang telah dilakukan.
6. **BAB VI : KESIMPULAN DAN SARAN**
   Bab ini berisi kesimpulan dari hasil uj coba yang telah dilakukan dan saran mengenai hal-hal yang masih bisa dikembangkan.

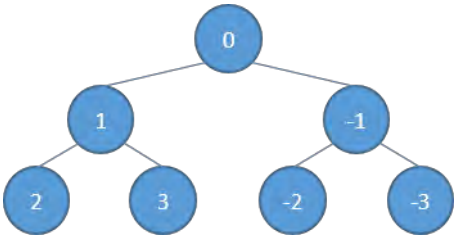
BAB II
DASAR TEORI

Pada bab ini dijelaskan mengenai permasalahan *disjoint subtrees* dan dasar teori yang digunakan dalam pengerjaan penelitian ini.

2.1 Deskripsi Permasalahan SPOJ Klasik *Disjoint Subtrees*

Diberikan suatu bangunan yang memiliki N ruangan dan N–1 koridor. Di ruangan ke-i terdapat seseorang yang akan memberikan atau merampas permen sejumlah B_i. Dimana i bernilai antara nol hingga N–1. Setiap koridor menghubungkan dua ruangan. Setiap ruangan terhubung ke ruangan yang lain dengan tepat satu jalur.

Dua orang, Alpa dan Shed, berkeliling bangunan. Mereka diperbolehkan untuk memulai proses berkeliling bangunan dari ruangan yang mana saja. Bila salah satu dari mereka memasuki ruangan dimana terdapat seseorang yang akan memberikan B_i permen, maka dia akan berinteraksi dan menerima B_i permen. Namun, bila dia memasuki suatu ruangan dimana terdapat orang yang akan merampas B_i permen, maka dia akan berinteraksi dan memberikan B_i permen. Alpa dan Shed mungkin saja memiliki kurang dari nol permen, artinya mereka berhutang kepada para perampas permen dari ruangan yang mereka lewati. Alpa harus berinteraksi dengan K_1 orang di dalam bangunan. Shed harus berinteraksi dengan K_2 orang di dalam bangunan. Mereka diperbolehkan untuk mengunjungi suatu ruangan lebih dari satu kali untuk menuju ruangan yang mereka inginkan dengan interaksi pada ruangan tersebut tetap hanya pada saat pertama kali memasuki ruangan tersebut.



Gambar 2.1 *Tree* Contoh Permasalahan *Disjoint Subtrees*

Tujuan permasalahan ini adalah memaksimalkan perbedaan antara yang didapat Alpa dan Shed, total yang didapat Alpa dikurang total yang didapat Shed.

Sebagai contoh, diberikan bangunan seperti pada Gambar 2.1, terdapat tujuh ruangan dengan enam koridor. Dengan setiap ruangan terdapat satu nilai, nilai positif berarti terdapat orang yang akan memberikan permen, sebaliknya, nilai negatif berarti terdapat orang yang akan merampas permen. Bila nilai K_1 dan K_2 adalah tiga, maka hasil akhirnya adalah 12. Dimana Alpa mengumpulkan enam permen dan Shed berhutang enam permen.

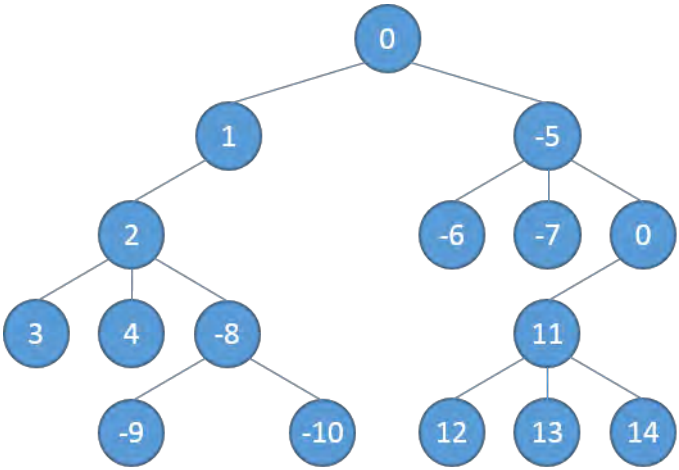
Bangunan pada permasalahan ini dapat dilihat sebagai sebuah *tree*. Dimana ruangan diwakili oleh sebuah *vertex* dan koridor diwakili oleh sebuah *edge*. Ruangan-ruangan yang dilalui oleh Alpa bisa dilihat sebagai himpunan A dan ruangan-ruangan yang dilalui oleh Shed bisa dilihat sebagai himpunan S. Nilai himpunan A adalah total permen yang didapat oleh Alpa. Nilai himpunan S adalah total permen yang didapat oleh Shed. Jumlah anggota himpunan adalah jumlah *vertex* dari himpunan tersebut.

Permasalahan ini adalah suatu permasalahan optimasi, dimana tujuan dari permasalahan ini adalah mencari nilai terbaik dari semua kemungkinan yang ada. Pendekatan pemrograman dinamis dapat menyelesaikan permasalahan ini karena permasalahan ini memiliki kriteria submasalah optimal dan submasalah tumpang tindih.

Dengan pendekatan pemrograman dinamis, penghitungan dimulai dari setiap *leaf* dan naik hingga mencapai *root*. Pada setiap *vertex* akan disimpan nilai-nilai submasalah pada *vertex* tersebut yang akan digunakan pada *vertex*-*vertex ancestor*-nya.

2.2 Analisa Submasalah Optimal pada Permasalahan SPOJ Klasik *Disjoint Subtrees*

Pada subbab ini dijelaskan mengenai submasalah-submasalah yang jawabannya dapat membangun jawaban akhir. Diberikan *tree* seperti yang diperlihatkan oleh Gambar 2.2 dengan ketentuan K_1 bernilai empat dan K_2 bernilai tiga.

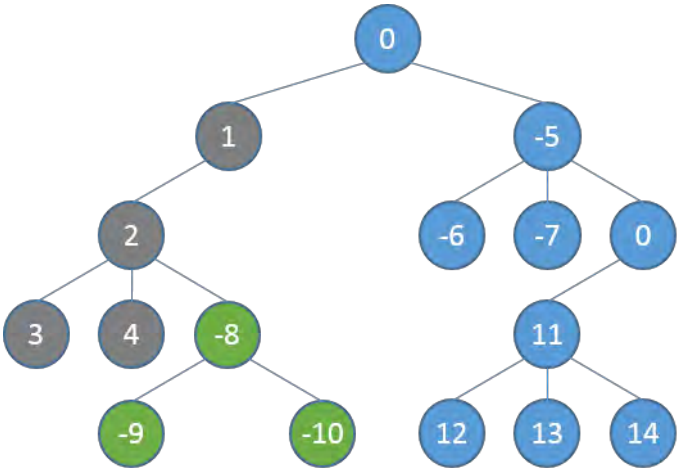


Gambar 2.2 *Tree* Analisa Tipe Hasil Akhir

Hasil akhir yang dicari adalah nilai terbaik dari nilai himpunan A dikurang nilai himpunan S. Dari semua kemungkinan jawaban yang mungkin, secara garis besar terdapat tiga tipe hasil akhir.

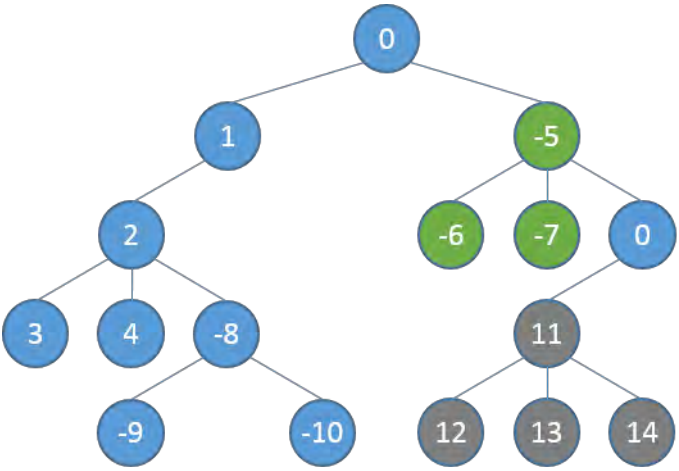
Tipe pertama adalah ketika salah satu himpunan berada pada jalur *root* menuju himpunan yang lainnya dan pada jalur himpunan ini menuju himpunan yang lainnya tidak terdapat *vertex* yang

memisahkan kedua himpunan. Seperti yang terlihat pada Gambar 2.3, dimana himpunan A berada diantara jalur *root* menuju himpunan S dan tidak ada *vertex* yang memisahkan kedua himpunan. Nilai hasil akhir dari kandidat ini adalah 37, dimana himpunan A bernilai sepuluh dan himpunan S bernilai -27. Nilai ini diperoleh ketika perhitungan telah mencapai *vertex* dengan nilai satu.

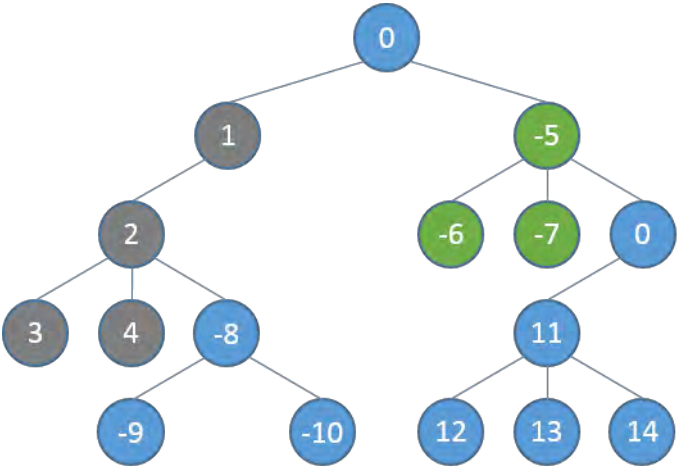


Gambar 2.3 Tipe Satu Hasil Akhir Permasalahan *Disjoint Subtrees*

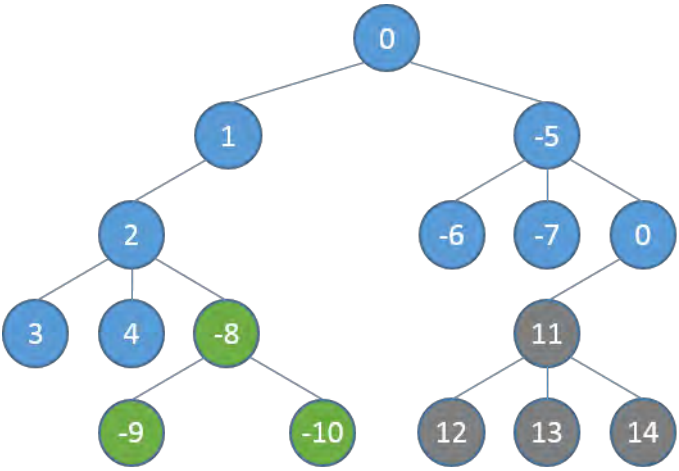
Tipe kedua adalah ketika salah satu himpunan berada pada jalur *root* menuju himpunan yang lainnya dan pada jalur dari himpunan ini menuju himpunan yang lainnya terdapat *vertex* memisahkan kedua himpunan. Seperti yang terlihat pada Gambar 2.4, dimana himpunan S berada diantara jalur *root* menuju himpunan A dan kedua himpunan dipisahkan oleh *vertex* dengan nilai nol. Nilai hasil akhir dari kandidat ini adalah 68, dimana himpunan A bernilai lima puluh dan himpunan S bernilai -18. Nilai ini diperoleh ketika perhitungan telah mencapai *vertex* dengan nilai minus lima.



Gambar 2.4 Tipe Dua Hasil Akhir Permasalahan *Disjoint Subtrees*



Gambar 2.5 Tipe Tiga Hasil Akhir Permasalahan *Disjoint Subtrees* (a)



Gambar 2.6 Tipe Tiga Hasil Akhir Permasalahan *Disjoint Subtrees* (b)

Tipe ketiga adalah ketika himpunan A dan S memiliki *common ancestor* yang berada di luar kedua himpunan. Seperti yang terlihat pada Gambar 2.5 dan Gambar 2.6, dimana pada kedua gambar tersebut terlihat bahwa dalam hal ini *root* menjadi *common ancestor* kedua himpunan. Kandidat pada Gambar 2.5, bernilai 28. Sedangkan kandidat pada Gambar 2.6, bernilai 77.

Dari setiap kandidat hasil akhir pada permasalahan yang digambarkan pada Gambar 2.2, semua berawal dari kondisi dimana kedua himpunan belum memiliki anggota. Dari kondisi awal ini hingga mencapai kondisi yang diminta oleh tujuan, ada beberapa nilai yang harus dicatat pada setiap *vertex* sehingga perhitungan menjadi efisien. Terdapat tujuh submasalah dimana nilai dari setiap submasalah pada setiap *vertex* akan digunakan pada *ancestor*-nya untuk perhitungan yang efisien.

Submasalah A pada suatu *vertex* V adalah nilai terbesar pada *vertex* V ketika himpunan A telah memiliki N anggota dan himpunan S belum memiliki anggota, dimana N kurang dari sama dengan K_1 dan *vertex* V termasuk ke dalam himpunan A.

Submasalah B pada suatu *vertex* V adalah nilai terkecil pada *vertex* V ketika himpunan A belum memiliki anggota dan himpunan S telah memiliki N anggota, dimana N kurang dari sama dengan K_2 dan *vertex* V termasuk ke dalam himpunan S.

Submasalah C pada suatu *vertex* V adalah nilai terbesar hingga *vertex* V ketika himpunan A telah memiliki K_1 anggota dan himpunan S belum memiliki anggota, dimana *vertex* V tidak harus termasuk ke dalam himpunan A.

Submasalah D pada suatu *vertex* V adalah nilai terkecil hingga *vertex* V ketika himpunan A belum memiliki anggota dan himpunan S telah memiliki K_2 anggota, dimana *vertex* V tidak harus termasuk ke dalam himpunan S.

Submasalah E pada suatu *vertex* V adalah nilai terbesar ketika himpunan A memiliki N anggota dan himpunan S telah memiliki K_2 anggota, dimana N kurang dari sama dengan K_1 dan *vertex* V termasuk anggota himpunan A.

Submasalah F pada suatu *vertex* V adalah nilai terbesar ketika himpunan A telah memiliki K_1 anggota dan himpunan S memiliki N anggota, dimana N kurang dari sama dengan K_2 dan *vertex* V termasuk anggota himpunan S.

Submasalah G pada suatu *vertex* V adalah nilai terbesar ketika himpunan A telah memiliki K_1 anggota dan himpunan S telah memiliki K_2 anggota. Dengan kata lain submasalah G adalah kandidat hasil akhir.

Hasil akhir dapat dibentuk oleh beberapa cara, seperti yang terlihat pada tipe-tipe hasil akhir. Tipe pertama dibentuk oleh submasalah E pada *vertex* dengan nilai satu, seperti yang diperlihatkan pada Gambar 2.3. Submasalah E dibentuk oleh penggabungan submasalah A dan submasalah D pada *vertex* dengan nilai dua. Submsalah D dibentuk oleh submasalah B.

Tipe kedua dibentuk oleh submasalah F pada *vertex* dengan nilai minus lima, seperti yang diperlihatkan pada Gambar 2.4. Dimana submasalah F itu sendiri dibentuk oleh penggabungan nilai submasalah B dan submasalah C pada *vertex* dengan nilai minus lima. Pada *vertex* dengan nilai nol yang merupakan *child* dari *vertex* dengan nilai minus lima, nilai submasalah C pada *vertex* tersebut bernilai sama besar dengan *child* dari *vertex* tersebut yang dibentuk oleh submasalah A pada *vertex* dengan nilai sebelas.

Tipe ketiga dibentuk oleh penggabungan submasalah C dan submasalah D, seperti yang diperlihatkan pada Gambar 2.5 dan Gambar 2.6.

Terlihat bahwa submasalah A akan membangun submasalah C. Submasalah B akan membangun submasalah D. Submasalah A dan submasalah D akan membangun submasalah E. Submasalah B dan submasalah C akan membangun submasalah F.Untuk submasalah G, dibangun oleh submasalah C dan submasalah D, submasalah E, atau submasalah F.

2.3 Definisi dan Notasi

Berikut adalah notasi yang akan digunakan pada penelitian ini.

1. $DP_{(V,\ K,\ X)}$ adalah notasi untuk nilai dari submasalah K, dimana K bernilai A, B, E, atau F, pada *vertex* V ketika N bernilai X.
2. $DP2_{(V,\ K)}$ adalah notasi untuk nilai dari submasalah K, dimana K bernilai C, D, atau G, pada *vertex* V.
3. $DPc_{(V,\ K,\ X)}$ adalah notasi untuk nilai dari submasalah K, dimana K bernilai A, B, E, atau F, pada gabungan seluruh *children vertex* V ketika N bernilai X.
4. $DP2c_{(V,\ K,\ X)}$ adalah notasi untuk nilai dari submasalah K, dimana K bernilai C, D, atau G, pada gabungan seluruh *children vertex* V.

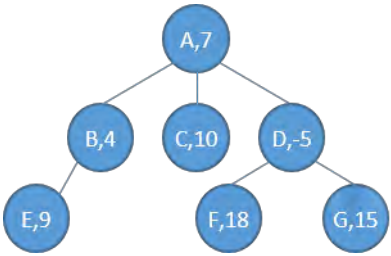
5. $B_{(V)}$ adalah jumlah permen yang ada pada *vertex* V. Nilai positif berarti Alpa/Sed akan menerima $B_{(V)}$ permen. Sebaliknya, nilai negatif berarti Alpa/Sed akan dirampas $B_{(V)}$ permen.

2.4 Pemodelan Relasi Rekurens pada Permasalahan SPOJ Klasik *Disjoint Subtrees*

Untuk *tree* seperti yang terlihat pada Gambar 2.7, $DP_{(A,\ A,\ 1)}$ bernilai tujuh, yaitu bobot pada *vertex* A itu sendiri. Sedangkan $DP_{(A,\ A,\ 3)}$ bernilai 21, tersusun oleh *vertex* A, B, C. Dapat disimpulkan bahwa secara umum relasi rekurens untuk submasalah A seperti yang terlihat pada persamaan (1). Sedangkan untuk submasalah B, secara garis besar sama dengan submasalah A. Relasi rekurens dari submasalah B seperti yang terlihat pada persamaan (2). $DP_{(V,\ A,\ 0)}$ dan $DP_{(V,\ B,\ 0)}$ bernilai 0 pada setiap *vertex*. Nilai $DPc_{(A,\ A,\ N)}$ untuk *tree* seperti pada Gambar 2.7, dapat dilihat pada Tabel 2.1.

$$DP_{(V,A,N)} = DPc_{(V,A,N-1)} + B_{(V)} \quad (1)$$

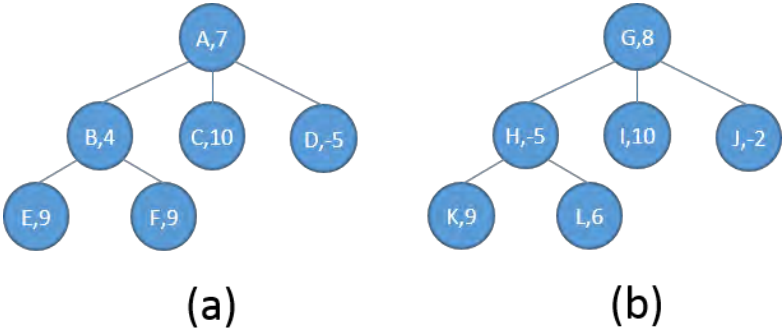
$$DP_{(V,B,N)} = DPc_{(V,B,N-1)} - B_{(V)}` \quad (2)$$



Gambar 2.7 *Tree* Ilustrasi Submasalah A

Tabel 2.1 Nilai Submasalah A pada *Tree* seperti pada Gambar 2.7

N	$DPc_{(A, A, N)}$	*Vertex* penyusun	$DP_{(A, A, N+1)}$
0	0	-	7
1	10	C	17
2	14	B, C	21
3	28	D, F, G	35
4	38	C, D, F, G	45
5	42	B, C, D , F,G	52
6	51	B, C, D, E, F, G	61



Gambar 2.8 *Tree* Ilustrasi Submasalah D

Untuk *tree* seperti yang terlihat pada Gambar 2.8, bila K_1 bernilai tiga, maka $DP2_{(A, C)}$ bernilai 22, sedangkan $DP2_{(G, C)}$ bernilai 16. $DP2_{(A, C)}$ terbentuk dari $DP2_{(B, C)}$, sedangkan $DP2_{(G, C)}$ terbentuk dari $DP_{(G, A, K1)}$. Maka dapat ditarik kesimpulan bahwa secara umum relasi rekurens untuk submasalah C dapat dilihat pada persamaan (3). Sedangkan untuk submasalah D, secara garis besar sama dengan submasalah C, dapat dilihat pada persamaan (4).

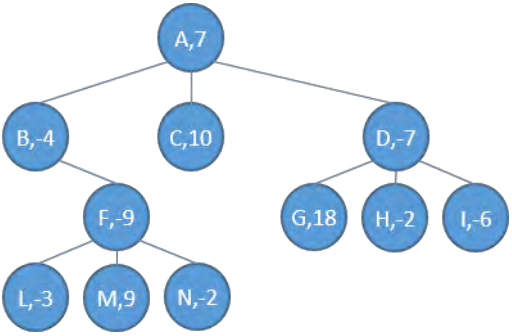
$$DP2_{(V,C)} = max\left(DP_{(V,A,K_1)}, DP2c_{(V,C)}\right) \quad (3)$$

$$DP2_{(V,D)} = max\left(DP_{(V,B,K_2)}, DP2c_{(V,D)}\right) \quad (4)$$

Untuk *tree* seperti yang terlihat pada Gambar 2.9, bila K_2 bernilai tiga, maka nilai $DP_{(A,\ E,\ 1)}$ adalah 23, terdiri dari *vertex* A, B, F, dan L. Sedangkan nilai $DP_{(A,\ E,\ 3)}$ bernilai 28, terdiri dari *vertex* A, B, C, D, H, dan I. Nilai $DP_{(A,\ E,\ 1)}$ didapat dari menjumlahkan nilai *vertex* A dengan $DP2_{(B,\ D)}$ sedangkan nilai $DP_{(A,\ E,\ 3)}$ didapat dari menjumlahkan nilai *vertex* A dengan $DPc_{(V,\ E,\ 2)}$, yaitu $DP_{(B,\ A,\ 1)}$, $DP_{(C,\ A,\ 1)}$, dan $DP2_{(D,\ D)}$. Nilai submasalah E pada *children vertex* A dengan N bernilai nol hingga tiga seperti yang terlihat pada Tabel 2.2. Bisa disimpulkan bahwa relasi rekurens dari submasalah E seperti pada persamaan (5). Untuk relasi rekurens submasalah F seperti yang terlihat pada persamaan (6).

$$DP_{(V,E,N)} = \begin{cases} DPc_{(V,E,N-1)} + B_{(V)}, N > 0 \\ DP2_{(V,D)}, N = 0 \end{cases} \quad (5)$$

$$DP_{(V,F,N)} = \begin{cases} DPc_{(V,F,N-1)} - B_{(V)}, N > 0 \\ DP2_{(V,C)}, N = 0 \end{cases} \quad (6)$$



Gambar 2.9 *Tree* Ilustrasi Submasalah E

Tabel 2.2 Nilai Submasalah E pada *Tree* seperti pada Gambar 2.9

N	$DPc_{(A, E, N)}$	*Vertex* penyusun
0	16	B, F, L
1	26	B, C, F, L
2	21	B, C, D, H, I
3	30	B, C, D, F, H, I

Pada subbab 2.2 telah disinggung mengenai beberapa kemungkinan hasil akhir yang merupakan $DP2_{(V, G)}$. Terlihat dari semua kemungkinan hasil akhir yang valid bahwa persamaan (7) merupakan relasi rekurens dari submasalah G secara umum.

$$DP2_{(V,G)} = max\big(DP2c_{(V,G)}, DP_{(V,E,N)}, DP_{(V,F,N)}, case3\big), C_1 = 1\ to\ CC, C_2 = 1\ to\ CC \quad (7)$$

$$case3 = max_{(C_1 \neq C_2)}\big(DP_{(C_1,A,K_1)} + DP_{(C_2,B,K_2)}\big) \quad (8)$$

Dapat dilihat dari persamaan (1) hingga (7), bahwa setiap persamaan yang dilakukan pada suatu *vertex* melibatkan penggabungan nilai-nilai submasalah pada *children* dari *vertex* tersebut.

Bila diberikan *vertex* V yang memiliki dua *children* dan *subtree* yang memiliki *root* pada *child* pertama memiliki a *vertex* sedangkan *subtree* yang memiliki *root* pada *child* kedua memiliki b *vertex*. ka adalah nilai terbesar diantara K_1 dan a. kb adalah nilai terbesar diantara K_2 dan b. Maka proses penggabungan nilai-nilai pada *children* tersebut berdasarkan submasalahnya adalah sebagai berikut:

Submasalah A dan B hasil penggabungan nilai-nilai pada *children* didapat dengan persamaan seperti yang terlihat pada persamaan (9) dan (10).

$$DPc_{(V,A,x)} = max_{(X_1 + X_2 = X)}\left(DP_{(C_1,A,X_1)} + DP_{(C_2,A,X_2)}\right), \quad X_1 = 0\ to\ ka, X_2 = 0\ to\ kb \tag{9}$$

$$DPc_{(V,B,x)} = max_{(X_1 + X_2 = X)}\left(DP_{(C_1,B,X_1)} + DP_{(C_2,B,X_2)}\right), \quad X_1 = 0\ to\ ka, X_2 = 0\ to\ kb \tag{10}$$

Submasalah C dan D hasil penggabungan nilai-nilai pada *children* didapat dengan persamaan seperti yang terlihat pada persamaan (11) dan (12). Dimana persamaan tersebut mencari nilai submasalah C terbaik dari semua *children*.

$$DP2c_{(V,C)} = max\left(DP2_{(C_x,C)}\right), x = 1\ to\ 2 \tag{11}$$

$$DP2c_{(V,D)} = max\left(DP2_{(C_x,D)}\right), x = 1\ to\ 2 \tag{12}$$

Untuk submasalah E dan F hasil penggabungan nilai-nilai pada *children* didapat dengan persamaan seperti yang terlihat pada persamaan (13) dan (14).

$$DPc_{(V,E,x)} = max_{(X_1 + X_2 = X)}\begin{pmatrix} DP_{(C_1,A,X_1)} + DP_{(C_2,E,X_2)}, \\ DP_{(C_1,E,X_1)} + DP_{(C_2,A,X_2)} \end{pmatrix}, \quad X_1 = 0\ to\ ka, X_2 = 0\ to\ kb \tag{13}$$

$$DPc_{(V,F,x)} = max_{(X_1 + X_2 = X)}\begin{pmatrix} DP_{(C_1,B,X_1)} + DP_{(C_2,F,X_2)}, \\ DP_{(C_1,F,X_1)} + DP_{(C_2,B,X_2)} \end{pmatrix}, \quad X_1 = 0\ to\ ka, X_2 = 0\ to\ kb \tag{14}$$

Untuk submasalah G hasil penggabungan nilai-nilai pada *children* didapat dengan persamaan seperti yang terlihat pada persamaan (15) dan (16).

$$DP2c_{(V,G)} = max\left(DP2_{(C_x,G)}\right), x = 1\ to\ 2 \quad (15)$$

$$case3 = max\begin{pmatrix} DP2_{(C_1,C)} + DP2_{(C_2,D)}, \\ DP2_{(C_1,D)} + DP2_{(C_2,C)} \end{pmatrix} \quad (16)$$

Persamaan-persamaan tersebut berlaku ketika jumlah child adalah dua. Namun bila suatu *vertex* memiliki *children* dalam jumlah yang besar, maka dengan pendekatan naif, proses penggabungan nilai-nilai ini bisa menjadi sangat memakan waktu karena mencari kombinasi dari semua kemungkinan. Dua pendekatan untuk menyelesaikan permasalahan yang timbul akibat dari jumlah *children* pada suatu *vertex* yang lebih dari dua adalah dengan menggunakan pemrograman dinamis dalam menggabungkan nilai-nilai submasalah pada *children* atau dengan mengkonversi *arbitrary tree* data masukan menjadi *left child right sibling tree* agar dapat dipastikan bahwa setiap *vertex* memiliki *children* tidak lebih dari dua.

2.5 Pendekatan Pertama dalam Menyelesaikan Permasalahan SPOJ Klasik Disjoint Subtrees

Pendekatan ini menggunakan metode pemrograman dinamis dalam menggabungkan nilai-nilai submasalah ketika suatu *vertex* memiliki lebih dari dua *children*.

Penggabungan nilai-nilai submasalah pada *children* suatu *vertex* dengan pendekatan ini dilakukan dengan menggabungkan dua *child* yang berdekatan. Bila *vertex* V memiliki c *children*, nilai-nilai submasalah terbaik dari seluruh *children* didapat dari penggabungan nilai-nilai submasalah terbaik dari gabungan antara c – 1 *children* pertama dengan *child* terakhir. Nilai-nilai terbaik dari c – 1 children pertama didapat dari gabungan antara c – 2 *children* pertama dengan child ke-c – 1. Hingga penggabungan nilai antara child pertama dengan child kedua.

Submasalah untuk proses penggabungan adalah nilai terbaik hingga child ke-c untuk submasalah S dengan nilai N adalah n, $DP3_{(c,\ s,\ n)}$. Dengan relasi rekurens seperti pada persamaan (9) hingga (16). Setelah nilai gabungan dari seluruh *children* didapat, maka persamaan (1) hingga (8) dapat dilakukan.

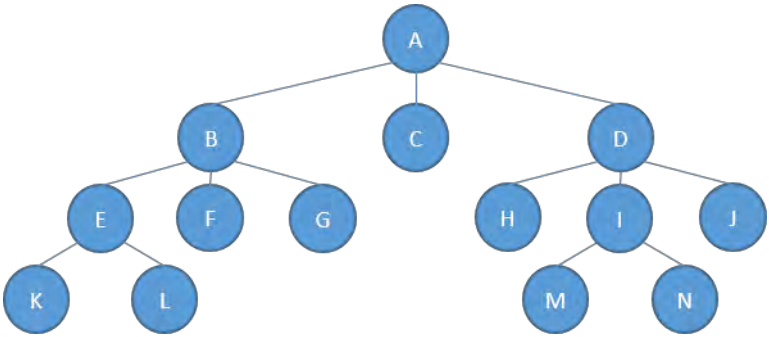
Secara keseluruhan maka proses penyelesaian permasalahan *disjoint subtrees* dengan pendekatan ini dilakukan dari *leaf* hingga *root* menggunakan DFS. Setiap *vertex* menyimpan nilai-nilai setiap submasalah dari semua *vertex* yang berada pada *subtree* yang memiliki *root* pada *vertex* tersebut.

Secara lebih detil proses penyelesaian permasalahan *disjoint subtrees* dengan pendekatan ini dilakukan dengan tiga langkah, yaitu:

1. Untuk setiap *vertex*, dimulai dari *leaf* hingga sampai pada *root*, lakukan dua hal ini.
    a. Nilai-nilai submasalah pada *children* digabungkan dengan menggunakan metode pemrograman dinamis.
    b. Nilai-nilai submasalah untuk *vertex* ini didapat dengan persamaan (1) hingga (7) seperti yang telah dibahas pada subbab 2.4.
2. Hasil akhir terletak pada nilai $DP_{(root,\ G)}$.

2.6 Pendekatan Kedua dalam Menyelesaikan Permasalahan SPOJ Klasik Disjoint Subtrees

Pendekatan ini menggunakan mengubah *arbitrary tree* data masukan menjadi *left child right sibling tree* sehingga setiap *vertex* memiliki maksimal dua *children*. Setiap *vertex* menyimpan nilai submasalah terbaik dari keturunannya dan juga dari *sibling* setelahnya. Secara tidak langsung proses penggabungan nilai submasalah pada *children* dilakukan secara bertahap pada setiap *vertex*.

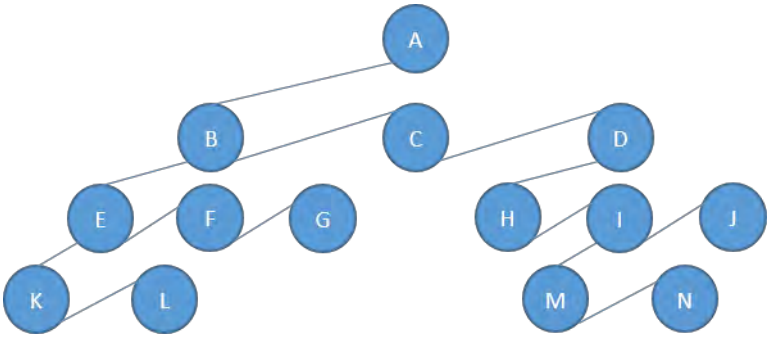


Gambar 2.10 Contoh *Arbitrary Tree* Data Masukan

LCRS tree adalah *tree* dengan maksimal *edge* keluar dari setiap *vertex*-nya adalah dua. Dimana *edge* keluar pertama menghubungkan *vertex* tersebut dengan *child* pertamanya dan *edge* keluar kedua menghubungkan *vertex* tersebut dengan *sibling* berikutnya.

Pembuatan *LCRS tree* terbagi menjadi tiga bagian, penyediaan *vertex* sebanyak *vertex* dari *arbitrary tree* data masukan, inisialisasi *edge* keluar dari setiap *vertex*, dan pembangunan hubungan *edge* keluar dari setiap *vertex*.

Proses pembangunan hubungan *edge* keluar dari setiap *vertex* adalah untuk setiap *vertex*, bila memiliki *child*, maka hubungkan *edge* keluar pertama dari *vertex* tersebut dengan *child* pertama, lalu hubungkan *edge* keluar kedua dari *child* ke-i menuju *child* ke-i+1, dari *child* pertama hingga *child* sebelum terakhir *vertex* tersebut.



Gambar 2.11 Hasil konversi *Arbitrary Tree* Menjadi *LCRS Tree*

Gambar 2.11 adalah contoh dari hasil konversi *arbitrary tree* yang terlihat seperti pada Gambar 2.10. Tidak ada *vertex* yang memiliki *edge* keluar lebih dari dua, dengan begitu proses penggabungan nilai-nilai pada *children* dilakukan secara bertahap. Setiap *vertex* menyimpan nilai-nilai gabungan antara nilai-nilai pada *subtree* yang memiliki *root* pada *vertex* tersebut dan nilai-nilai pada *sibling*-nya. Sehingga *child* pertama dari suatu *vertex* menyimpan nilai-nilai gabungan dari semua *children vertex* tersebut.

Secara keseluruhan maka proses penyelesaian permasalahan *disjoint subtrees* dengan pendekatan ini dilakukan dari *leaf* hingga *root* menggunakan DFS. Setiap *vertex* menyimpan nilai-nilai setiap submasalah dari semua *vertex* yang berada pada *subtree* yang memiliki *root* pada *vertex* tersebut dan *subtree* yang memiliki *root* pada *vertex*-*vertex sibling* berikutnya.

Secara lebih detil proses penyelesaian permasalahan *disjoint subtrees* dengan pendekatan ini dilakukan dengan tiga langkah, yaitu:

1. Konversi *arbitrary tree* data masukan menjadi *LCRS tree*.
2. Untuk setiap *vertex*, dimulai dari *leaf* hingga sampai pada *root*, lakukan dua hal ini.

a. Persamaan (1) hingga (7) dilakukan dalam proses mendapatkan nilai dari penggabungan *vertex* tersebut dengan *edge* keluar pertama, yaitu dengan *child*-nya. Hasil ini disimpan di dalam memori sementara.
b. Nilai $DP2_{(V,G)}$ diperbaharui dengan nilai terbesar diantara $DP2c_{(V,G)}$, submasalah E ketika N bernilai K_1 dari memori sementara, dan submasalah F ketika N bernilai K_2 dari memori sementara.
c. Hasil pada memori sementara dengan nilai-nilai yang ada pada *edge* keluar kedua, yaitu dengan sibling-nya digabungkan dengan persamaan (9) hingga (15).
d. Nilai $DP2_{(V,G)}$ diperbaharui dengan nilai terbesar diantara $DP2_{(V,G)}$ dan persamaan (16).

3. Hasil akhir terletak pada nilai $DP_{(root, G)}$.

2.7 Pembuatan Data Generator Untuk Uji Coba

Pembuatan data generator bertujuan untuk membuat kasus uji dalam rangka melihat performa dari algoritma yang dibuat pada penelitian ini.

Data Generator memberikan kasus uji yang disesuaikan dengan data masukan yang dijelaskan pada subbab 4.2.1. Akan terbentuk sebuah *tree* yang seimbang yang *children* dan kedalaman *tree* sesuai dengan masukan yang diberikan pada data generator.

Proses pembuatan kasus uji yang dilakukan oleh data generator seperti yang terlihat pada Gambar 2.12. Baris keenam menuliskan jumlah uji coba yaitu sebanyak satu kali. Baris ketujuh menuliskan nilai jumlah *vertex*, K_1, dan K_2. Baris kedelapan hingga kesembilan menuliskan nilai bobot setiap *vertex*. Baris

kesebelas hingga keempat belas adalah menuliskan *tree* sesuai dengan format data masukan.

createTestCase()	
1.	$branch \leftarrow jumlah\ children\ pada\ setiap\ vertex$
2.	$deep \leftarrow kedalaman\ tree$
3.	$K_1 \leftarrow nilai\ K_1$
4.	$K_2 \leftarrow nilai\ K_2$
5.	$N \leftarrow (branch^{deep+1} - 1)\ /\ (branch - 1)$
6.	$print$ "1"
7.	$print\ N, K_1, K_2$
8.	$for\ i \leftarrow 0\ to\ N - 1$
9.	$print\ randomNumber()$
10.	$child \leftarrow 1$
11.	$for\ i \leftarrow 0\ to\ (N\ /\ branch) - 1$
12.	$for\ j \leftarrow 0\ to\ branch - 1$
13.	$print\ i\ child$
14.	$child = child + 1$

Gambar 2.12 ***Pseudocode Data Generator***

BAB III
DESAIN

Pada bab ini dijelaskan mengenai desain algoritma dan struktur data yang digunakan pada penelitian ini.

3.1 Desain Umum Sistem

Pada subbab ini dijelaskan mengenai gambaran secara umum dari kedua pendekatan.

Program menerima masukan berupa banyak data uji. Untuk setiap data uji terdiri atas tiga bagian, yaitu: bagian pertama berisi jumlah *vertex* pada *arbitrary tree* masukan, K_1, dan K_2. Pada bagian kedua berisi bobot pada setiap *vertex*, dan bagian terakhir berisi *edge* yang menghubungkan *vertex-vertex* yang membentuk *tree*. Setelah menerima masukan, maka masukan tersebut diolah dan hasilnya ditampilkan di layar. Secara garis besar seperti yang terlihat pada Gambar 3.1. Fungsi *solveProblem* bergantung kepada pendekatan yang digunakan. Di dalam fungsi *solveProblem* terdapat fungsi-fungsi *preprocess* dan fungsi penyelesaian permasalahan.

3.2 Desain Algoritma

Pada subbab ini dijelaskan fungsi-fungsi yang terdapat pada sistem dari kedua pendekatan yang telah dibahas pada subbab 2.5 dan subbab 2.6.

`Main()`	
1.	$tc \leftarrow jumlah\ data\ uji$
2.	$for\ i \leftarrow 0\ to\ tc - 1$
3.	$readInput()$
4.	$solveProblem()$
5.	$writeOutput()$

Gambar 3.1 Alur Penyelesaian Secara Garis Besar

solvProblem()	
1.	$initMemo()$
2.	$countVerexInSubtree()$
3.	$findAnswer()$

Gambar 3.2 Alur Penyelesaian Menggunakan Algoritma Pendekatan Pertama

Input: node: vertex Output: -	
1.	$for\ i \leftarrow 0\ to\ number\ of\ children\ of\ vertex\ node$
2.	$findAnswer(tree_{(node,i)})$
3.	$initMemoTemp(node)$
4.	$merging(node)$
5.	$fillMemo(node)$

Gambar 3.3 *Pseudocode* Fungsi findAnswer

3.2.1 Pendekatan Pertama

Pada subbab ini dijelaskan desain dari fungsi-fungsi dalam penyelesaian permasalahan *Disjoint Subtrees* dengan pendekatan pertama seperti yang dijelaskan pada subbab 2.5.

Agar algoritma dengan pendekatan pertama dapat berjalan dengan baik, maka fungsi preprocess harus dilakukan terlebih dahulu. Seperti yang terlihat pada Gambar 3.2, terdapat fungsi *initMemo* dan *countVertex*. Setelah kedua fungsi *preprocess* tersebut dijalankan maka fungsi *findAnswer* dijalankan.

Perhitungan untuk mendapatkan jawaban akhir dilakukan dari *leaf* hingga kembali ke *root*, fungsi ini dapat dipecah menjadi tiga bagian yang dilakukan pada setipa *vertex* seperti yang terlihat pada Gambar 3.3, yaitu:

1. Inisialisasi memori sementara. Bertujuan agar proses pencarian nilai submasalah berjalan dengan benar.

2. Penggabungan nilai submasalah *children*. Proses ini seperti yang telah dijelaskan pada subbab 2.5.

Perhitungan nilai submasalah pada *vertex* itu sendiri. Proses ini seperti yang telah dijelaskan pada subbab 2.4.

3.2.1.1 Desain Fungsi Preprocess

Fungsi *preprocess* merupakan fungsi yang bertujuan agar algoritma yang menyelesaikan permasalahan dapat berjalan dengan benar dan efisien. Dalam algortima dengan pendekatan pertama, fungsi *preprocess* terbagi menjadi dua, inisialisasi memori tempat menyimpan nilai-nilai submasalah dan perhitungan jumlah *vertex* pada setiap *subtree* yang terdapat dalam *tree*.

Pada Gambar 3.4 terlihat *pseudocode* bagaimana memori penyimpanan nilai submasalah pada setiap *vertex* pada *tree* data masukan diinisialisasi dengan nilai minus tak terhingga.

`Input:`
 –
 `Output:`
 –	

1.	$for\ i \leftarrow 0\ to\ number\ of\ vertex$
2.	$DP2_{(i,C)} \leftarrow -INF$
3.	$DP2_{(i,D)} \leftarrow -INF$
4.	$DP2_{(i,G)} \leftarrow -INF$
5.	$for\ j \leftarrow 0\ to\ K_1$
6.	$DP_{(i,A,j)} \leftarrow -INF$
7.	$DP_{(i,E,j)} \leftarrow -INF$
8.	$for\ j \leftarrow 0\ to\ K_2$
9.	$DP_{(i,B,j)} \leftarrow -INF$
10.	$DP_{(i,F,j)} \leftarrow -INF$

Gambar 3.4 *Pseudocode* Fungsi initMemo

Pada Gambar 3.5 terlihat bahwa proses penghitungan jumlah *vertex* untuk *substree* pada *tree* data masukan menggunakan pendekatan DFS. *Subtree* yang memiliki *vertex* lebih besar dari nilai terbesar diantara K_1 dan K_2 dianggap memiliki *vertex* dengan jumlah nilai terbesar diantara K_1 dan K_2. Ini bertujuan agar perhitungan menjadi lebih efisien.

3.2.1.2 Desain Fungsi Penyelesaian Permasalahan Disjoint Subtrees

Pada subbab ini dijelaskan secara lebih detil mengenai fungsi-fungsi penyusun fungsi *findAnswer* seperti yang terlihat pada Gambar 3.3.

Fungsi *initMemoTemp* seperti yang terlihat pada Gambar 3.6, bertugas mempersiapkan memori tempat penyimpanan tambahan untuk menyimpan hasil penggabungan nilai-nilai submasalah dari setiap *children*. Sebelum digunakan oleh fungsi *merging* agar dapat berjalan dengan benar.

Input: node: vertex root dari subtree limit: nilai terbesar antara K_1 dan K_2 Output: -	
1.	$sumNode_{(node)} \leftarrow 1$
2.	$for\ i \leftarrow 0\ to\ number\ of\ children\ of\ vertex\ node$
3.	$countVertex(tree_{(node,i)})$
4.	$sumNode_{(node)} \leftarrow sumNode_{(node)} + sumNode_{(tree_{(node,i)})}$
5.	$if\ sumNode_{(node)} > limit$
6.	$sumNode_{(node)} = limit$

Gambar 3.5 *Pseudocode* Fungsi countVertex

Input: Node: vertex root of the subtree Output: -	
1.	$tDP2_{(C)} \leftarrow -INF$
2.	$tDP2_{(D)} \leftarrow -INF$
3.	$tDP2_{(G)} \leftarrow -INF$
4.	$for\ j \leftarrow 0\ to\ sumNode_{(node)}$
5.	$tDP_{(A,j)} \leftarrow -INF$
6.	$tDP_{(B,j)} \leftarrow -INF$
7.	$tDP_{(E,j)} \leftarrow -INF$
8.	$tDP_{(F,j)} \leftarrow -INF$
9.	$tDP_{(A,0)} \leftarrow 0$
10.	$tDP_{(i,B,0)} \leftarrow 0$

Gambar 3.6 *Pseudocode* Fungsi initMemoTemp

Input: Node: vertex root of the subtree Output: -	
1.	$sumVertexPrev \leftarrow 1$
2.	$limit \leftarrow max(K_1, K_2)$
3.	$for\ i \leftarrow 0\ to\ number\ of\ children\ of\ vertex\ node$
4.	$shiftMemoTemp(sumVertexPrev)$
5.	$fillMemoTemp(sumVertexPrev, tree_{(node,i)})$
6.	$if\ sumVertexPrev > limit$
7.	$sumVertexPrev = limit$
8.	$else$
9.	$sumVertexPrev \leftarrow sumVertexPrev + sumNode_{(tree_{(node,i)})}$

Gambar 3.7 *Pseudocode* Fungsi merging

Fungsi *merging* seperti yang terlihat pada Gambar 3.7, bertugas untuk menggabungkan nilai-nilai submasalah dari setiap *children*. Perhitungan dimulai dari *child* pertama hingga terakhir, dengan

menggabungkan nilai-nilai dari dua *child* yang bersebelahan. Terbagi menjadi dua fungsi, yaitu:

1. Menyimpan nilai submasalah pada memori sementara ke dalam memori sementara sebelumnya.
2. Menggabungkan nilai submasalah pada memori sementara sebelumnya dengan memori yang menyimpan nilai submasalah *child* ke-I dan menyimpannya ke dalam memori sementara.

Fungsi *shiftMemoTemp* seperti yang terlihat pada Gambar 3.8, menyalin nilai-nilai pada memori sementara ke dalam memori sementara sebelumnya dan menginisialisasi memori sementara.

Input: sumVertex: nilai terkecil diantara jumlah vertex dari semua subtree sebelumnya dengan nilai terbesar diantara K_1 dan K_2. Output: -	
1.	$ttDP2_{(C)} \leftarrow tDP2_{(C)}$
2.	$tDP2_{(C)} \leftarrow -INF$
3.	$ttDP2_{(D)} \leftarrow tDP2_{(D)}$
4.	$tDP2_{(D)} \leftarrow -INF$
5.	$ttDP2_{(G)} \leftarrow tDP2_{(G)}$
6.	$tDP2_{(G)} \leftarrow -INF$
7.	$for\ i \leftarrow 0\ to\ sumVertex$
8.	$ttDP_{(A,i)} \leftarrow tDP_{(A,i)}$
9.	$tDP_{(A,j)} \leftarrow -INF$
10.	$ttDP_{(B,i)} \leftarrow tDP_{(B,i)}$
11.	$tDP_{(B,j)} \leftarrow -INF$
12.	$ttDP_{(E,i)} \leftarrow tDP_{(E,i)}$
13.	$tDP_{(E,j)} \leftarrow -INF$
14.	$ttDP_{(F,i)} \leftarrow tDP_{(F,i)}$
15.	$tDP_{(F,j)} \leftarrow -INF$

Gambar 3.8 *Pseudocode* Fungsi shiftMemoTemp

Fungsi *fillMemoTemp* seperti yang terlihat pada Gambar 3.9, adalah proses penggabungan nilai pada memori sementara yang berisi nilai submasalah gabungan *child* pertama hingga ke-i-1 dengan *child* ke-i. Persamaan (9), (10), (13), dan (14) dilakukan pada fungsi ini.

Fungsi *fillMemo* seperti yang terlihat pada Gambar 3.10, bertugas mengisi memori yang menyimpan nilai submasalah pada *vertex* node. Persamaan (1), (2), (5), (6) dilakukan pada baris empat hingga tujuh. Baris delapan dan Sembilan dilakukan persamaan (3) dan (4). Pada baris 14 hingga 20 dilakukan persamaan (7) dan (8).

Input: sumVertex: nilai terkecil diantara jumlah vertex dari semua subtree sebelumnya dengan nilai terbesar diantara K_1 dan K_2. child: vertex berikutnya yang nilai tabelnya akan digabungkan. Output: -	
1.	$for\ i \leftarrow 0\ to\ sumVertex$
2.	$for\ j \leftarrow 0\ to\ sumNode_{(child)}$
3.	$tDP_{(A,i+j)} \leftarrow max\left(tDP_{(A,i+j)}, DP_{(child,A,i)} + ttDP_{(A,j)}\right)$
4.	$tDP_{(B,i+j)} \leftarrow max\left(tDP_{(B,i+j)}, DP_{(child,B,i)} + ttDP_{(B,j)}\right)$
5.	$tDP_{(E,i+j)} \leftarrow max\left(tDP_{(E,i+j)}, max\left(DP_{(child,E,i)} + tDP_{(A,j)}, DP_{(child,A,i)} + tDP_{(E,j)}\right)\right)$
6.	$tDP_{(E,i+j)} \leftarrow max\left(tDP_{(E,i+j)}, max\left(DP_{(child,E,i)} + tDP_{(A,j)}, DP_{(child,A,i)} + tDP_{(E,j)}\right)\right)$

Gambar 3.9 *Pseudocode* fungsi fillMemoTemp

Input: node: jumlah vertex dari semua subtree sebelumnya. Output: -	
1.	$DP_{(node,A,0)} \leftarrow 0$
2.	$DP_{(node,B,0)} \leftarrow 0$
3.	$for\ i \leftarrow 0\ to\ sumNode_{(node)}$
4.	$DP_{(node,A,i+1)} \leftarrow tDP_{(A,i)} + Weight_{(node)}$
5.	$DP_{(node,B,i+1)} \leftarrow tDP_{(B,i)} - Weight_{(node)}$
6.	$DP_{(node,E,i+1)} \leftarrow tDP_{(E,i)} + Weight_{(node)}$
7.	$DP_{(node,F,i+1)} \leftarrow tDP_{(F,i)} - Weight_{(node)}$
8.	$DP2_{(node,C)} \leftarrow max(DP_{(node,A,K_1)}, tDP_{(F,0)})$
9.	$DP2_{(node,D)} \leftarrow max(DP_{(node,B,K_2)}, tDP_{(E,0)})$
10.	$tans \leftarrow max(DP_{(node,E,K_1)}, DP_{(node,F,K_2)})$
11.	$if\ number\ of\ children\ of\ vertex\ node > 1$
12.	$maxa \leftarrow DP2_{(tree_{(node,0)},C)}$
13.	$maxb \leftarrow DP2_{(tree_{(node,0)},D)}$
14.	$for\ i \leftarrow 0\ to\ number\ of\ children\ of\ vertex\ node$
15.	$temp \leftarrow max\left(maxa + DP2_{(tree_{(node,i)},D)}, maxb + DP2_{(tree_{(node,i)},C)}\right)$
16.	$tans \leftarrow max(tans, temp)$
17.	$maxa \leftarrow max\left(maxa, DP2_{(tree_{(node,i)},C)}\right)$
18.	$maxb \leftarrow max\left(maxb, DP2_{(tree_{(node,i)},D)}\right)$
19.	$DP2_{(node,G)} \leftarrow max\left(DP2_{(node,G)}, DP2_{(tree_{(node,i)},G)}\right)$
20.	$DP2_{(node,G)} \leftarrow max(DP2_{(node,G)}, tans)$

Gambar 3.10 *Pseudocode* fungsi fillMemo

solvProblem()	
1.	$initMemo()$
2.	$constructLCRSTree()$
3.	$countVerexInSubtree()$
4.	$findAnswer()$

Gambar 3.11 Alur Penyelesaian Menggunakan Algoritma Pendekatan Kedua

Input: node: vertex posisi sekarang Output: -	
1.	$for\ i \leftarrow 0\ to\ 2$
2.	$if\ LCRSTree_{(i,node)} \neq N$
3.	$findAnswer(LCRSTree_{(i,node)})$
4.	$initMemoTemp(node)$
5.	$processChild(node, LCRSTree_{(0,node)})$
6.	$processSibling(node, LCRSTree_{(0,node)}, LCRSTree_{(1,node)})$

Gambar 3.12 *Pseudocode* Fungsi findAnswer

3.2.2 Pendekatan Kedua

Pada subbab ini dijelaskan desain dari fungsi-fungsi dalam penyelesaian permasalahan *Disjoint Subtrees* dengan pendekatan kedua seperti yang telah dijelaskan pada subbab 2.6.

Agar algoritma dengan pendekatan kedua dapat berjalan dengan baik, maka fungsi *preprocess* harus dilakukan terlebih dahulu. Seperti yang terlihat pada Gambar 3.11, terdapat fungsi *initMemo*, *constructLCRSTree*, dan *countVertexInSubstree*. Setelah ketiga fungsi preprocess tersebut dijalankan maka fungsi *findAnswer* dijalankan.

Perhitungan untuk mendapatkan jawaban akhir dilakukan dari *leaf* hingga kembali ke *root*, fungsi ini dapat dipecah menjadi tiga bagian seperti yang terlihat pada Gambar 3.12 yang juga dibahas pada subbab 2.6, yaitu:

1. Inisialisasi memo temporari. Bertujuan agar proses pencarian nilai submasalah berjalan dengan benar.
2. Menghitung nilai submasalah *vertex* tersebut bersama dengan *children*-nya.
3. Menggabungkan nilai submasalah pada *vertex* itu sendiri dengan nilai pada *sibling*-nya.

3.2.2.1 Desain Fungsi-Fungsi Preprocess

Fungsi *preprocess* merupakan fungsi tambahan untuk mempersiapkan data masukan agar dapat diproses oleh algoritma pemrograman dinamis dengan metode LCRS terlebih dahulu. Fungsi preprocess ini terbagi menjadi tiga. Yaitu:

1. Inisialisasi memori tempat menyimpan nilai-nilai submasalah dan *LCRS tree*.
2. Membangun *LCRS tree* dari *arbitrary tree* data masukan.
3. Menghitung jumlah *vertex* setiap *subtree* pada LCRS *tree*.

Fungsi inisialisasi *LCRS tree* seperti yang terlihat pada Gambar 3.13, *vertex* pada *LCRS tree* diisi dengan nilai N yang menandakan tidak ada *vertex* yang dihubungkan olehnya. Fungsi ini digunakan agar *LCRS tree* sesuai dengan *arbitrary tree* data masukan ketika dibangun pada fungsi *constructLCRSTree*.

`Input:`
 `-`
 `Output:`
 `-`	

1.	$for\ i \leftarrow 0\ to\ 2$
2.	$for\ j \leftarrow 0\ to\ number\ of\ vertex$
3.	$BTree_{(i,j)} \leftarrow N$

Gambar 3.13 *Pseudocode* Fungsi initLCRSTree

Input: – Output: –	
1.	$for\ i \leftarrow 0\ to\ number\ of\ vertex$
2.	$DP2_{(i,C)} \leftarrow -INF$
3.	$DP2_{(i,D)} \leftarrow -INF$
4.	$DP2_{(i,G)} \leftarrow -INF$
5.	$for\ j \leftarrow 0\ to\ K_1$
6.	$DP_{(i,A,j)} \leftarrow -INF$
7.	$DP_{(i,E,j)} \leftarrow -INF$
8.	$for\ j \leftarrow 0\ to\ K_2$
9.	$DP_{(i,B,j)} \leftarrow -INF$
10.	$DP_{(i,F,j)} \leftarrow -INF$
11.	initLCRSTree()

Gambar 3.14 *Pseudocode* Fungsi initMemo

Input: – Output: –	
1.	$for\ i \leftarrow 0\ to\ number\ of\ vertex$
2.	$if\ children\ of\ vertex_i > 0$
3.	$BTree_{(0,i)} \leftarrow tree_{(i,0)}$
4.	$for\ j \leftarrow 0\ to\ number\ of\ children\ of\ vertex_i$
5.	$BTree_{(1,tree_{(i,j)})} \leftarrow tree_{(i,j+1)}$

Gambar 3.15 *Pseudocode* Fungsi constructLCRSTree

Fungsi *initMemo* seperti yang terlihat pada Gambar 3.14, fungsi ini menginisialisasi nilai yang ada pada memori penyimpanan nilai submasalah dan memanggil fungsi *initLCRSTree*.

Setelah proses inisialisasi selesai, maka fungsi *constructLCRSTree* seperti yang terlihat pada Gambar 3.15, dipanggil. Fungsi ini membangun *LCRS Tree* seperti yang dibahas pada subbab 2.6.

Input: node: vertex yang akan dihitung jumlah keturunannya. Output: -	
1.	$sumNode_{(node)} \leftarrow 1$
2.	$for\ i \leftarrow 0\ to\ 2$
3.	$if\ LCRSTree_{(i,node)} \neq N$
4.	$countVertex(LCRSTree_{(i,node)})$
5.	$sumNode_{(node)} \leftarrow sumNode_{(node)} + sumNode_{(LCRSTree_{(i,node)})}$
5.	$if\ sumNode_{(node)} > limit$
6.	$sumNode_{(node)} = limit$

Gambar 3.16 *Pseudocode* Fungsi countVertex

Fungsi *countVertex* seperti yang terlihat pada Gambar 3.16, digunakan untuk mendapatkan jumlah *vertex* pada *subtree* yang terdapat pada *LCRS tree*. *Subtree* yang memiliki *vertex* lebih besar dari nilai terbesar diantara K_1 dan K_2 dianggap memiliki *vertex* dengan jumlah nilai terbesar diantara K_1 dan K_2. Ini bertujuan agar perhitungan menjadi lebih efisien.

3.2.2.2 Desain Fungsi Penyelesaian Permasalahan Disjoint Subtrees

Pada subbab ini dijelaskan secara lebih detil mengenai fungsi-fungsi penyusun fungsi *findAnswer* seperti yang terlihat pada Gambar 3.12.

Fungsi *initMemoTemp* seperti yang terlihat pada Gambar 3.17, berfungsi untuk menginisialisasi memori penyimpanan nilai submasalah sementara yang digunakan dalam fungsi *processChild*.

Input: Node: vertex root of the subtree Output: -	
1.	$tDP2_{(C)} \leftarrow -INF$
2.	$tDP2_{(D)} \leftarrow -INF$
3.	$tDP2_{(G)} \leftarrow -INF$
4.	$for\ j \leftarrow 0\ to\ sumNode_{(node)}$
5.	$tDP_{(A,j)} \leftarrow -INF$
6.	$tDP_{(B,j)} \leftarrow -INF$
7.	$tDP_{(E,j)} \leftarrow -INF$
8.	$tDP_{(F,j)} \leftarrow -INF$

Gambar 3.17 *Pseudocode* Fungsi initMemoTemp

Input: Node: vertex root of the subtree Output: -	
1.	$if\ child \neq N$
2.	$for\ i \leftarrow 0\ to\ 2$
3.	$tDP_{(i,0)} \leftarrow 0$
4.	$for\ i \leftarrow 0\ to\ sumNode_{(child)}$
5.	$tDP_{(A,i+1)} \leftarrow DP_{(child,A,i)} + Weight_{(node)}$
6.	$tDP_{(B,i+1)} \leftarrow DP_{(child,B,i)} - Weight_{(node)}$
7.	$tDP_{(E,i+1)} \leftarrow DP_{(child,E,i)} + Weight_{(node)}$
8.	$tDP_{(F,i+1)} \leftarrow DP_{(child,F,i)} + Weight_{(node)}$
9.	$if\ K_1 \leq sumNode_{(child)} + 1$
10.	$tDP2_{(C)} \leftarrow max\left(tDP2_{(C)}, max\left(tDP_{(A,K_1)}, DP2_{(child,C)}\right)\right)$
11.	$tDP_{2(D)} \leftarrow max\left(tDP2_{(D)}, max\left(tDP_{(B,K_2)}, DP_{(child,D)}\right)\right)$
12.	$else$
13.	$tDP_{(A,0)} \leftarrow 0$
14.	$tDP_{(B,0)} \leftarrow 0$

Gambar 3.18 *Pseudocode* Fungsi processChild

Fungsi *processChild* seperti yang terlihat pada Gambar 3.18, bertugas mendapatkan nilai submasalah dari *vertex* tersebut dan menyimpannya ke dalam memori penyimpanan nilai submasalah sementara. Terdapat dua kemungkinan, yaitu *vertex* tersebut memiliki *child*, maka baris ke dua hingga sebelas akan dijalankan dan persamaan (1) hingga (7) digunakan, sebaliknya, bila tidak memiliki *child*, maka baris 13 hingga 21 yang akan dijalankan.

Fungsi *processSibling* seperti yang terlihat pada Gambar 3.19, bertugas untuk menggabungkan nilai submasalah pada *vertex* tersebut dengan nilai submasalah gabungan pada *vertex-vertex* yang merupakan *sibling* dari *vertex* tersebut. Persamaan (9) hingga (16) digunakan bila *vertex* tersebut memiliki *sibling*. Bila tidak memiliki *sibling*, maka nilai pada memori penyimpanan submasalah sementara disalin ke dalam memori penyimpanan nilai submasalah *vertex* tersebut.

	`Input:`
	`  node: vertex`
	`  child: vertex anak kiri dari node`
	`  sibling: vertex anak kanan dari node`
	`Output:`
	`  -`

1.	$if\ sibling\ \neq N$
2.	$DP2_{(node,G)} \leftarrow max\left(DP2_{(node,G)}, max\left(tDP2_{(C)} + DP2_{(sibling,D)}, tDP2_{(D)} + DP2_{(sibling,C)}\right)\right)$
3.	$for\ i \leftarrow 0\ to\ sumNode_{(child)} + 1$
4.	$for\ j \leftarrow 0\ to\ sumNode_{(sibling)}$
5.	$DP_{(node,A,i+j)} \leftarrow max\left(DP_{(node,A,i+j)}, tDP_{(A,i)} + DP_{(sibling,A,j)}\right)$
6.	$DP_{(node,B,i+j)} \leftarrow max\left(DP_{(node,B,i+j)}, tDP_{(B,i)} + DP_{(sibling,B,j)}\right)$
7.	$DP_{(node,E,i+j)} \leftarrow max\left(DP_{(node,E,i+j)}, max\left(tDP_{(E,i)} + DP_{(sibling,A,j)}, tDP_{(A,i)} + DP_{(sibling,E,j)}\right)\right)$

8.	$DP_{(node,F,i+j)} \leftarrow max\left(DP_{(node,F,i+j)}, max\left(tDP_{(F,i)} + DP_{(sibling,B,j)}, tDP_{(B,i)} + DP_{(sibling,F,j)}\right)\right)$
9.	$DP2_{(node,C)} \leftarrow \max\left(DP_{(node,A,K_1)}, max\left(tDP2_{(child,C)}, tDP2_{(sibling,C)}\right)\right)$
10.	$DP2_{(node,D)} \leftarrow \max\left(DP_{(node,B,K_2)}, max\left(tDP2_{(child,D)}, tDP2_{(sibling,D)}\right)\right)$
11.	$else$
12.	$for\ i \leftarrow 0\ to\ sumNode_{(node)}$
13.	$DP_{(node,A,i)} \leftarrow tDP_{(A,i)}$
14.	$DP_{(node,B,i)} \leftarrow tDP_{(B,i)}$
15.	$DP_{(node,E,i)} \leftarrow tDP_{(E,i)}$
16.	$DP_{(node,F,i)} \leftarrow tDP_{(F,i)}$
17.	$DP2_{(node,C)} \leftarrow tDP2_{(C)}$
18.	$DP2_{(node,D)} \leftarrow tDP2_{(D)}$
19.	$DP2_{(node,G)} \leftarrow tDP2_{(G)}$

Gambar 3.19 *Pseudocode* Fungsi processSibling

BAB IV
IMPLEMENTASI

Pada bab ini dijelaskan mengenai implementasi dari desain algoritma penyelesaian permasalahan SPOJ Klasik *Disjoint Subtrees*.

4.1 Lingkungan Implementasi

Lingkungan implementasi dalam pembuatan penelitian ini meliputi perangkat keras dan perangkat lunak yang digunakan untuk melakukan proses pendekatan algoritma pemrograman dinamis untuk permasalahan *Disjoint Subtrees* adalah sebagai berikut:

1. Perangkat Keras.
   Prosesor Intel(R) Core(TM) i5-3230M CPU @ 2.650GHz 2.59GHz RAM 4.00 GB.
   64-bit Operating System, x64-based processor.
2. Perangkat Lunak.
   Sistem operasi Windows Embedded 8.1 Indusry Pro.
   Integrated Development Environtment Code::Blocks 13.12.

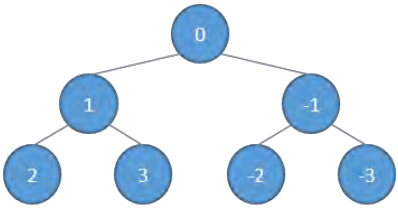
4.2 Rancangan Data

Pada subbab ini dijelaskan mengenai desain data masukan yang diperlukan untuk melakukan proses algoritma, dan data keluaran yang dihasilkan oleh program.

4.2.1 Data Masukan

Data masukan adalah data yang akan diproses oleh program sebagai masukan menggunakan algoritma pemrograman dinamis dalam penelitian ini.

Baris	Isi
1	7 3 3
2	0 1 -1 2 3 -2 -3
3	0 1
4	0 2
5	1 3
6	1 4
7	2 5
8	2 6



Gambar 4.1 Contoh Data Masukan Beserta Representasi Tree-*nya*

Data masukan diawali dengan jumlah kasus uji, untuk setiap kasus uji, formatnya adalah seperti Gambar 4.1. baris pertama berisi tiga bilangan, jumlah *vertex*, K_1, dan K_2. Baris kedua berisi bobot setiap *vertex*. N-1 baris berikutnya berisi dua bilangan, yaitu dua *vertex* yang dihubungkan oleh suatu *edge*.

4.2.2 Data Keluaran

Data keluaran yang dihasilkan oleh program hanya satu nilai, yaitu nilai terbaik dari nilai yang didapat oleh Alpa yang telah berinteraksi dengan K_1 ruangan dikurangi oleh nilai yang didapat oleh Shed yang telah berinteraksi dengan K_2 ruangan. Data keluaran bernilai -1 bila nilai ketika Alpa berinteraksi dengan K_1 ruangan dan Shed berinteraksi dengan K_2 ruangan tidak mungkin dicapai.

4.3 Implementasi Proses Algoritma

Pada subbab ini akan dijelaskan tentang implementasi proses algoritma secara keseluruhan berdasarkan desain yang telah dijelaskan pada bab 3.

4.3.1 Header-Header yang Diperlukan

Implementasi algoritma dengan pendekatan pemrograman dinamis untuk menyelesaikan permasalahan *Disjoint Subtrees*, baik untuk pendekatan pemrograman dinamis ataupun pendekatan *LCRS* dalam proses penggabungan nilai tabel memoisasi setiap *child* suatu *vertex,* membutuhkan tiga *header*, yaitu *cstdio*, *vector*, dan *algorithm*, seperti yang terlihat pada Kode Sumber 4.1.

Header cstdio berisi modul untuk menerima masukan dan memberikan keluaran. *Header vector* berisi struktur data yang digunakan untuk menyimpan *tree*. *Header algorithm* berisi modul untuk mendapatkan nilai terbesar dari dua buah nilai.

4.3.2 Variabel Global

Variabel global digunakan untuk memudahkan dalam mengakses data yang digunakan lintas fungsi, seperti yang terlihat pada Kode Sumber 4.2.

```
1.  #include <cstdio>
2.  #include <vector>
3.  #include <algorithm>
```

Kode Sumber 4.1 *Header* yang Diperlukan

```
1.  using namespace std;
2.  const int SIZE = 202;
3.  const int INF = 1000000000;
4.  const int bound = -20000000;
5.  int N, V[2], weight[SIZE];
6.  vector<int> tree[SIZE];
```

Kode Sumber 4.2 *Variabel Global*

4.3.3 Implementasi Desain Algoritma dengan Pendekatan Pertama

Sekilas properti dan fungsi yang dimiliki kelas yang mengimplementasikan pendekatan pertama seperti yang terlihat pada Kode Sumber 4.3.

```
1.  class KaykayDP {
2.  public:
3.   void readInput() {...}
4.   void solveProblem(int root) {...}
5.   void writeOutput() {...}
6.
7.  private:
8.   int sumNode[SIZE];
9.   int ans;
10.  int mem[4][SIZE][SIZE];
11.  int tmem[4][SIZE];
12.  int ttmem[4][SIZE];
13.
14.  void initMemo() {...}
15.  void countVertex(int node, int limit) {...}
16.  void initMemoTemp(int node) {...}
17.  void shiftMemoTemp(int sumVertex) {...}
18.  void fillMemoTemp(int sumVertex, int child) {...}
19.  void merging(int node) {...}
20.  void fillMemo(int node) {...}
21.  void findAnswer(int node) {...}
22. };
```

Kode Sumber 4.3 Implementasi Kelas Algoritma dengan Pendekatan Pertama

4.3.3.1 Properti Kelas dengan Public Modifier

Pada subbab ini dijelaskan secara lebih detil mengenai properti kelas seperti yang terlihat pada Kode Sumber 4.3 dengan akses publik.

4.3.3.1.1 Implementasi Fungsi readInput

Fungsi *readInput* seperti yang terlihat pada Kode Sumber 4.4, fungsi untuk menerima data masukan.

```
1.  void readInput() {
2.   scanf("%d %d %d", &N, &V[0], &V[1]);
3.
4.   for (int i = 0; i < N; i++)
5.    tree[i].clear();
6.
7.   for (int i = 0; i < N; i++)
8.    scanf("%d", &weight[i]);
9.
10.  int a, b;
11.  for (int i = 1; i < N; i++) {
12.   scanf("%d %d", &a, &b);
13.   tree[a].push_back(b);
14.  }
    }
```

Kode Sumber 4.4 Implementasi Fungsi readInput pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.1.2 Implementasi Fungsi solveProblem

Fungsi *solveProblem* seperti yang terlihat pada Kode Sumber 4.5, merupakan fungsi untuk menyelesaikan permasalahan *Disjoint Subtrees*.

```
1.  void solveProblem(int root) {
2.   initMemo();
3.   countVertex(root, max(V[0], V[1]);
4.   findAnswer(root);
    }
```

Kode Sumber 4.5 Implementasi Fungsi solveProblem pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.1.3 Implementasi Fungsi writeOutput

Fungsi *writeOutput* seperti yang terlihat pada Kode Sumber 4.6, digunakan untuk mencetak hasil akhir, yaitu *ans* atau $DP2_{(root,\ G)}$.

```
1.  void writeOutput() {
2.   if(N < V[0] + V[1] || ans < BOUND) printf("-1\n");
3.   else printf("%d\n", ans);
4.  }
```

Kode Sumber 4.6 Implementasi Fungsi writeOutput pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2 Properti Kelas dengan Private Modifier

Pada subbab ini dijelaskan secara lebih detil mengenai properti kelas seperti yang terlihat pada Kode Sumber 4.3 dengan akses privat.

4.3.3.2.1 Variabel Kelas

Seperti yang terlihat pada Kode Sumber 4.7, sumNode[x] menyimpan jumlah vertex dari subtree yang memiliki root vertex x. ans menyimpan nilai $DP2_{(node,\ G)}$. mem[x][y][z] menyimpan nilai $DP_{(y,\ x,\ z)}$. mem[2][y][0] menyimpan nilai $DP2_{(y,\ C)}$ sedangkan mem[3][y][0] menyimpan nilai $DP2_{(y,\ D)}$. tmem[x][y] menyimpan nilai $tDP_{(y,\ x)}$. tmem[2][0] menyimpan nilai $tDP2_{(C)}$ sedangkan tmem[3][0] menyimpan nilai $tDP2_{(D)}$.

```
1.  int sumNode[SIZE];
2.  int ans;
3.  int mem[4][SIZE][SIZE];
4.  int tmem[4][SIZE];
5.  int ttmem[4][SIZE];
```

Kode Sumber 4.7 Variabel Privat Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.2 Implementasi Fungsi initMemo

Implementasi fungsi initMemo seperti yang terlihat pada Kode Sumber 4.8, merupakan implementasi dari Gambar 3.4.

```
1.  void initMemo() {
2.   ans = -INF;
3.   for (int i = 0; i < 2; i++)
4.    for (int j = 0; j < N; j++)
5.     for (int k = 0; k <= V[i]; k++)
6.      mem[0 + i][j][k] = mem[2 + i][j][k] = -INF;
7.  }
```

Kode Sumber 4.8 Implementasi Fungsi initMemo pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.3 Implementasi Fungsi countVertex

Fungsi *countVertex* seperti yang terlihat pada Kode Sumber 4.9, merupakan implementasi dari Gambar 3.5.

```
1.  void countVertex(int node) {
2.   sumNode[node] = 1;
3.   for (int i = 0; i < tree[node].size(); i++) {
4.    countVertex(tree[node][i]);
5.    sumNode[node] += sumNode[tree[node][i]];
6.    if (sumNode[node] > limit) sumNode[node] = limit;
7.   }
8.  }
```

Kode Sumber 4.9 Implementasi Fungsi countVertex pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.4 Implementasi Fungsi findAnswer

Fungsi *findAnswer* seperti yang terlihat pada Kode Sumber 4.10, merupakan implementasi dari Gambar 3.3.

```
1.  void findAnswer(int node){
2.   for (int i = 0; i < tree[node].size(); i++)
3.    findAnswer(tree[node][i]);
4.
5.   initMemoTemp(node);
6.   merging(node);
7.   fillMemo(node);
8.  }
```

Kode Sumber 4.10 Implementasi Fungsi findAnswer pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.5 Implementasi Fungsi initMemoTemp

Fungsi *initMemoTemp* seperti yang terlihat pada Kode Sumber 4.11 merupakan implementasi dari Gambar 3.9.

```
1.  void initMemoTemp(int node) {
2.   for (int i = 0; i < 4; i++)
3.    for (int j = 0; j <= sumNode[node]; j++)
4.     tmem[i][j] = -INF;
5.   tmem[0][0] = tmem[1][0] = 0;
6.  }
```

Kode Sumber 4.11 Implementasi Fungsi initMemoTemp pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.6 Implementasi Fungsi merging

Fungsi *merging* seperti yang terlihat pada Kode Sumber 4.12, merupakan implementasi dari Gambar 3.7.

```
1.  void merging(int node) {
2.   int now = 1;
3.   for (int i = 0; i < tree[node].size(); i++) {
4.    int child = tree[node][i];
5.    shiftMemoTemp(now);
6.    fillMemoTemp(now, child);
7.    if (now + sumNode[child] > limit) now = limit;
8.    else now += sumNode[child]; } }
```

Kode Sumber 4.12 Implementasi Fungsi merging pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.7 Implementasi Fungsi fillMemoTemp

Fungsi *fillMemoTemp* seperti yang terlihat pada Kode Sumber 4.13, merupakan implementasi dari Gambar 3.9.

```
1.  void fillMemoTemp(int sumVertex, int child) {
2.   for (int j = 0; j <= sumVertex; j++)
3.    for (int k = 0; k <= sumNode[child]; k++) {
4.     tmem[0][j + k] = max(tmem[0][j + k],
    mem[0][child][k] + ttmem[0][j]);
5.     tmem[1][j + k] = max(tmem[1][j + k],
    mem[1][child][k] + ttmem[1][j]);
6.     tmem[2][j + k] = max(tmem[2][j + k],
    max(mem[2][child][k] + ttmem[0][j], mem[0][child][k] +
    ttmem[2][j]));
7.     tmem[3][j + k] = max(tmem[3][j + k],
    max(mem[3][child][k] + ttmem[1][j], mem[1][child][k] +
    ttmem[3][j]));
8.    }
9.  }
```

Kode Sumber 4.13 Implementasi Fungsi fillMemoTemp pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.8 Implementasi Fungsi shiftMemoTemp

Fungsi *shiftMemoTemp* seperti yang terlihat pada Kode Sumber 4.14, merupakan implementasi dari Gambar 3.8.

```
1.  void shiftMemoTemp(int sumVertex) {
2.   for (int j = 0; j < 4; j++)
3.    for (int k = 0; k <= sumVertex; k++) {
4.     ttmem[j][k] = tmem[j][k];
5.     tmem[j][k] = -INF;
6.    }
7.  }
```

Kode Sumber 4.14 Implementasi Fungsi shiftMemoTemp pada Kelas Algoritma dengan Pendekatan Pertama

4.3.3.2.9 Implementasi Fungsi fillMemo

Fungsi *fillMemo* seperti yang terlihat pada Kode Sumber 4.15, merupakan implementasi dari Gambar 3.10.

```
1.  void fillMemo(int node) {
2.   mem[0][node][0] = mem[1][node][0] = 0;
3.   for (int j = 0; j < sumNode[node]; j++) {
4.    mem[0][node][j + 1] = tmem[0][j] + weight[node];
5.    mem[1][node][j + 1] = tmem[1][j] - weight[node];
6.    mem[2][node][j + 1] = tmem[2][j] + weight[node];
7.    mem[3][node][j + 1] = tmem[3][j] - weight[node];
8.   }
9.   mem[2][node][0] = max(mem[1][node][V[1]],
     tmem[2][0]);
10.  mem[3][node][0] = max(mem[0][node][V[0]],
     tmem[3][0]);
11.
      int tans = max(mem[2][node][V[0]],
12.  mem[3][node][V[1]]), temp;
      if (tree[node].size() > 1) {
13.    int maxa = mem[2][tree[node][0]][0], maxb =
14.  mem[3][tree[node][0]][0];
       for (int i = 1; i < tree[node].size(); i++) {
15.     temp = max(maxa + mem[3][tree[node][i]][0], maxb +
16.  mem[2][tree[node][i]][0]);
        tans = max(tans, temp);
17.
18.     maxa = max(maxa, mem[2][tree[node][i]][0]);
19.     maxb = max(maxb, mem[3][tree[node][i]][0]);
20.    }
21.   }
22.
23.   if (ans < tans)
24.    ans = tans;
25.  }
```

Kode Sumber 4.15 Implementasi Fungsi fillMemo pada Kelas Algoritma dengan Pendekatan Pertama

4.3.4 Implementasi Desain Algoritma dengan Pendekatan Kedua

Sekilas properti dan fungsi yang dimiliki kelas yang mengimplementasikan pendekatan kedua seperti yang terlihat pada Kode Sumber 4.16.

```
1.  class KaykayLCRS {
2.  public:
3.   void readInput() {...}
4.   void solveProblem(int root) {...}
5.   void writeOutput() {...}
6.
7.  private:
8.   int ans;
9.   int sumNode[SIZE];
10.  int lcrstree[2][SIZE];
11.  int mem[4][SIZE][SIZE];
12.  int tmem[4][SIZE];
13.
14.  void initLCRSTree() {...}
15.  void initMemo() {...}
16.  void constructLCRSTree() {...}
17.  void countVertex(int node, int limit) {...}
18.  void initMemoTemp(int node) {...}
19.  void processChild(int node, int child) {...}
20.  void processSibling(int node, int child, int
     sibling) {...}
21.  void findAnswer(int node) {...}
22. };
```

Kode Sumber 4.16 Implementasi Kelas Algoritma dengan Pendekatan Kedua

4.3.4.1 Properti Kelas dengan Public Modifier

Pada subbab ini dijelaskan secara lebih detil mengenai properti kelas seperti yang terlihat pada Kode Sumber 4.16 yang memiliki akses publik.

4.3.4.1.1 Implementasi Fungsi readInput

Fungsi *readInput* seperti yang terlihat pada Kode Sumber 4.17, merupakan fungsi untuk menerima data masukan.

```
1.  void readInput() {
2.    scanf("%d %d %d", &N, &V[0], &V[1]);
3.
4.    for (int i = 0; i < N; i++)
5.     tree[i].clear();
6.
7.    for (int i = 0; i < N; i++)
8.     scanf("%d", &weight[i]);
9.
10.   int a, b;
11.   for (int i = 1; i < N; i++) {
12.    scanf("%d %d", &a, &b);
13.    tree[a].push_back(b);
14.   }
     }
```

Kode Sumber 4.17 Implementasi Fungsi readInput pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.1.2 Implementasi Fungsi solveProblem

Fungsi *solveProblem* seperti yang terlihat pada Kode Sumber 4.18, merupakan fungsi untuk menyelesaikan permasalahan *disjoint subtrees*.

```
1.  void solveProblem(int root) {
2.   initMemo()
3.   constructLCRSTree();
4.   countVertex(root);
5.   findAnswer(root);
    }
```

Kode Sumber 4.18 Implementasi Fungsi solveProblem pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.1.3 Implementasi Fungsi writeOutput

Fungsi *writeOutput* merupakan seperti yang terlihat pada Kode Sumber 4.19, merupakan fungsi untuk mencetak hasil akhir yaitu *ans* atau $DP2_{(root, G)}$.

```
1.  void writeOutput() {
2.   if (N < V[0] + V[1] || ans < BOUND) printf("-1\n");
3.   else printf("%d\n", ans);
4.  }
```

Kode Sumber 4.19 Implementasi fungsi writeOutput pada kelas Algoritma dengan pendekatan Kedua

4.3.4.2 Properti Kelas dengan Private Modifier

Pada subbab ini dijelaskan secara lebih detil mengenai properti kelas seperti yang terlihat pada Kode Sumber 4.16 yang memiliki akses privat.

4.3.4.2.1 Variabel Kelas

Variabel kelas seperti yang terlihat pada Kode Sumber 4.20, ans menyimpan nilai $DP2_{(node,\ G)}$. sumNode[x] menyimpan jumlah vertex dari subtree yang memiliki root pada vertex x. lcrstree[x][y] merupakan struktur data LCRS tree, x bernilai 0 menunjukan vertex child dari vertex y, sedangkan x bernilai 1 menunjukan sibling berikutnya dari vertex y. mem[x][y][z] menyimpan nilai $DP_{(y,\ x,\ z)}$. mem[2][y][0] menyimpan nilai $DP2_{(y,\ C)}$ sedangkan mem[3][y][0] menyimpan nilai $DP2_{(y,\ D)}$. tmem[x][y] menyimpan nilai $tDP_{(y,\ x)}$. tmem[2][0] menyimpan nilai $tDP2_{(C)}$ sedangkan tmem[3][0] menyimpan nilai $tDP2_{(D)}$.

```
1.  int ans;
2.  int sumNode[SIZE];
3.  int lcrstree [2][SIZE];
4.  int mem[4][SIZE][SIZE];
5.  int tmem[4][SIZE];
```

Kode Sumber 4.20 Variabel Kelas pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.2 Implementasi Fungsi initLCRSTree

Fungsi *initLCRSTree* seperti yang terlihat pada Kode Sumber 4.21, merupakan implementasi dari Gambar 3.13.

```
1.  void initLCRSTree() {
2.   for (int i = 0; i < 2; i++)
3.    for (int j = 0; j < N; j++)
4.     lcrstree[i][j] = N;
5.  }
```

Kode Sumber 4.21 Implementasi Fungsi initLCRSTree pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.3 Implementasi Fungsi initMemo

Fungsi *initMemo* seperti yang terlihat pada Kode Sumber 4.22, merupakan implementasi dari Gambar 3.14.

```
1.  void initMemo() {
2.   ans = -INF; initLCRSTree();
3.   for (int i = 0; i < 2; i++)
4.    for (int j = 0; j < N; j++)
5.     for (int k = 0; k < V[i]; k++)
6.      mem[0 + i][j][k] = mem[2 + i][j][k] = -INF; }
```

Kode Sumber 4.22 Implementasi Fungsi initMemo pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.4 Implementasi Fungsi countVertex

Fungsi *countVertex* seperti yang terlihat pada Kode Sumber 4.23, merupakan implementasi dari Gambar 3.16.

```
1.  void countVertex(int node) {
2.   sumNode[node] = 1;
3.   for (int i = 0; i < 2; i++)
4.    if (lcrstree [i][node] != N) {
5.     countVertex(lcrstree [i][node]);
6.     sumNode[node] += sumNode[lcrstree [i][node]];
7.    }
8.  }
```

Kode Sumber 4.23 Implementasi Fungsi countVertex pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.5 Implementasi Fungsi findAnswer

Fungsi *findAnswer* seperti yang terlihat pada Kode Sumber 4.24, merupakan implementasi dari *pseudocode* pada Gambar 3.12.

```
1.  void findAnswer(int node) {
2.   for (int i = 0; i < 2; i++)
3.    if (lcrstree [i][node] != N)
4.     findAnswer(lcrstree [i][node]);
5.   initMemoTemp(node);
6.   processChild(node, lcrstree [0][node]);
7.   processSibling(node, lcrstree [0][node], lcrstree
    [1][node]); }
```

Kode Sumber 4.24 Implementasi Fungsi findAnswer pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.6 Implementasi Fungsi initMemoTemp

Fungsi *initmemoTemp* seperti yang terlihat pada Kode Sumber 4.25, merupakan implementasi dari Gambar 3.17.

```
1.  void initMemoTemp(int node) {
2.   for (int i = 0; i < 4; i++)
3.    for (int j = 0, x = max(sumNode[node], max(V[0],
    V[1])); j <= x; j++)
4.     tmem[i][j] = -INF;
5.   for (int i = 0; i < 4; i++)
6.    tmem[i][V[0]] = tmem[i][V[1]] = -INF;
7.  }
```

Kode Sumber 4.25 Implementasi Fungsi initMemoTemp pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.7 Implementasi Fungsi processChild

Fungsi *processChild* seperti yang terlihat pada Kode Sumber 4.26, merupakan implementasi dari Gambar 3.18.

```
1.  void processChild(int node, int child) {
2.   if (child != N) {
3.    for (int i = 0; i < 2; i++)
4.     tmem[i][0] = 0;
5.    for (int i = 0; i < 4; i++)
6.     for (int j = 0; j <= sumNode[child]; j++)
7.      tmem[i][j + 1] = mem[i][child][j] + ((i%2 == 0) ?
    weight[node] : -weight[node]);
8.    for (int i = 0; i < 2; i++)
9.     if (V[i] <= sumNode[child] + 1)
10.     tmem[3 - i][0] = max(tmem[3 - i][0],
    max(tmem[i][V[i]], mem[3 - i][child][0]));
11.  } else {
12.    for (int i = 0; i < 2; i++)
13.     for (int j = 0; j < 2; j++) {
14.      if(j == 0) tmem[i][j] = 0;
15.      else tmem[i][j] = ((i == 0) ? weight[node] : -
    weight[node]);
16.     }
17.    for (int i = 0; i < 2; i++)
18.     if (V[i] == 1) tmem[3 - i][0] = max(tmem[3 -
    i][0], tmem[i][V[i]]);
19.   }
20.   ans = max(ans, max(tmem[2][V[0]], tmem[3][V[1]])); }
```

Kode Sumber 4.26 Implementasi fungsi processChild pada Kelas Algoritma dengan Pendekatan Kedua

4.3.4.2.8 Implementasi Fungsi processSibling

Fungsi *processSibling* seperti yang terlihat pada Kode Sumber 4.27, merupakan implementasi dari Gambar 3.19.

```
1.  void processSibling(int node, int child, int sibling)
    {
2.   if (sibling != N) {
3.     ans = max(ans, max(tmem[2][0] + mem[3][sibling][0],
    tmem[3][0] + mem[2][sibling][0]));
4.     for (int i = 0, x = sumNode[child] + 1; i <= x;
    i++)
         for (int j = 0; j <= sumNode[sibling]; j++) {
5.        mem[0][node][i + j] = max(mem[0][node][i + j],
6.   tmem[0][i] + mem[0][sibling][j]);
          mem[1][node][i + j] = max(mem[1][node][i + j],
7.   tmem[1][i] + mem[1][sibling][j]);
          mem[2][node][i + j] = max(mem[2][node][i + j],
8.   max(tmem[2][i] + mem[0][sibling][j], tmem[0][i] +
    mem[2][sibling][j]));
          mem[3][node][i + j] = max(mem[3][node][i + j],
9.   max(tmem[3][i] + mem[1][sibling][j], tmem[1][i] +
    mem[3][sibling][j]));
         }
10.  } else {
11.   for (int i = 0; i < 4; i++)
12.    for (int j = 0; j <= sumNode[node]; j++)
13.     mem[i][node][j] = tmem[i][j];
14.  }
15. }
16.
```

Kode Sumber 4.27 Implementasi Fungsi processSibling pada Kelas Algoritma dengan Pendekatan Kedua

BAB V
UJI COBA DAN EVALUASI

Pada bab ini dijelaskan uji coba dan evaluasi dari implementasi yang telah dijelaskan pada bab 4.

5.1 Lingkungan Uji Coba

Lingkungan uji coba dalam digunakan untuk algoritma penyelesaian permasalahan SPOJ Klasik 13948. *Disjoint Subtrees* adalah sebagai berikut :

1. Perangkat Keras
   Prosesor: Intel(R) Core(TM) i5-3230M CPU @ 2.650GHz 2.59GHz RAM 4.00GB.
   64-bit Operating System, x64-based processor.
2. Perangkat Lunak
   Windows Embedded 8.1 Indusry Pro
   Integrated Development Environtment Code::Blocks 13.12

5.2 Data Uji Coba

Data masukan untuk bahan uji coba berasal dari situs penilaian Daring yang bernama Sphere Online Judge (SPOJ) permasalahan klasik *Disjoint Subtrees*.

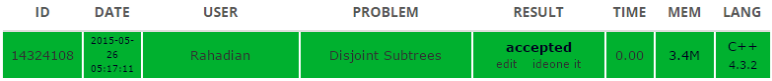
5.3 Skenario Uji Coba

Pada subbab ini dijelaskan mengenai berbagai uji coba kepada algoritma yang telah diimplementasikan.

5.3.1 Uji Coba Kebenaran

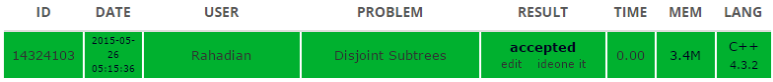
Kode sumber dari kedua pendekatan yang diberikan kepada situs penilaian daring mendapatkan umpan balik *Accepted*. Umpan balik *Accepted* menyatakan bahwa algoritma pada kode sumber

yang diberikan berhasil menyelesaikan permasalahan tersebut. Umpan balik dapat dilihat pada Gambar 5.1 untuk pendekatan Pertama dan Gambar 5.2 untuk pendekatan Kedua.



ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
14324108	2015-05-26 05:17:11	Rahadian	Disjoint Subtrees	accepted edit ideone it	0.00	3.4M	C++ 4.3.2

Gambar 5.1 Hasil Pengujian Pendekatan Pertama pada Situs SPOJ



ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
14324103	2015-05-26 05:15:36	Rahadian	Disjoint Subtrees	accepted edit ideone it	0.00	3.4M	C++ 4.3.2

Gambar 5.2 Hasil Pengujian Pendekatan Kedua pada Situs SPOJ

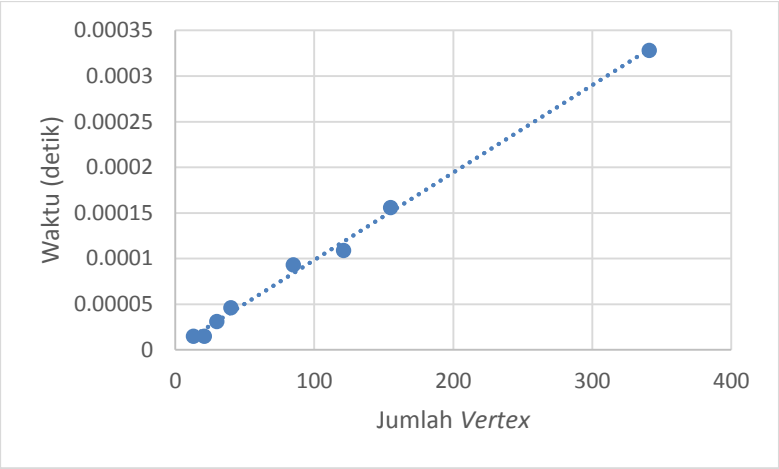
5.3.2 Uji Coba Kinerja

Pada subbab ini dijelaskan mengenai pengaruh beberapa variabel terhadap waktu.

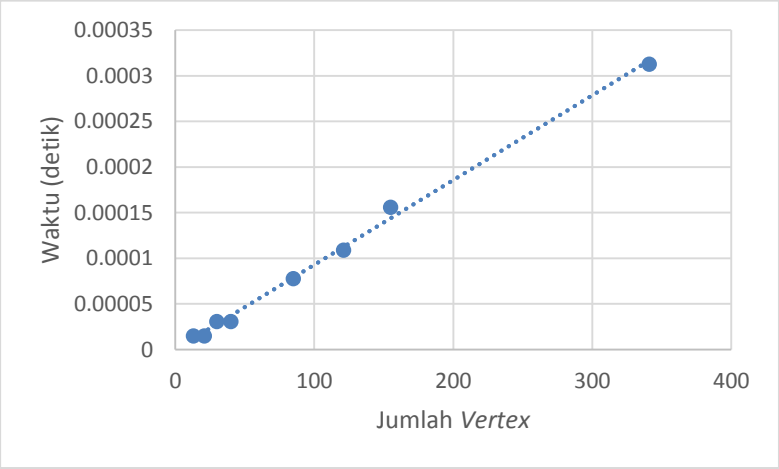
5.3.2.1 Pengaruh Jumlah Vertex Terhadap Waktu

Pada uji coba ini, nilai K_1 dan K_2 dibuat tetap dengan nilai tujuh dan tiga. Untuk jumlah *vertex* dibuat bertambah baik berdasarkan jumlah *children* pada setiap *vertex* maupun kedalaman dari *tree* masukan. Hasil uji coba ini seperti yang terlihat pada Gambar 5.3 untuk pendekatan pertama dan Gambar 5.4 untuk pendekatan kedua.

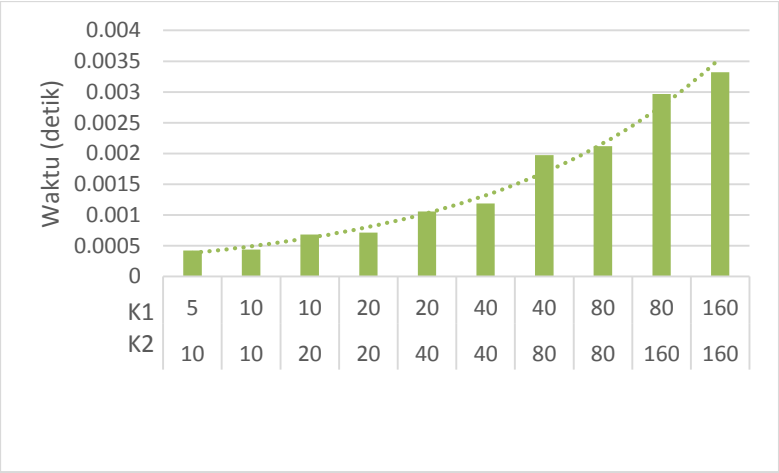
Terlihat dari kedua grafik bahwa pengaruh jumlah *vertex* terhadap waktu mendekati kurva linear.



Gambar 5.3 Grafik Hasil Uji Coba Pengaruh Banyaknya *Vertex* Terhadap Waktu untuk Pendekatan Pertama



Gambar 5.4 Grafik Hasil Uji Coba Pengaruh Banyaknya *Vertex* Terhadap Waktu untuk Pendekatan Kedua

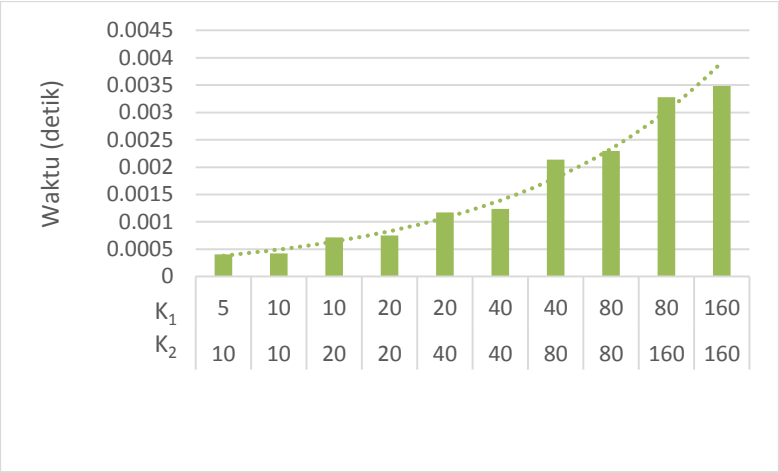


Gambar 5.5 Grafik Hasil Uji Coba Pengaruh Jumlah K_1 dan K_2 Terhadap Waktu untuk Pendekatan Pertama

5.3.2.2 Pengaruh Jumlah K_1 dan K_2 Terhadap Waktu

Pada uji coba ini, jumlah *vertex* dibuat tetap dengan jumlah 341. Setiap *vertex* memiliki empat *children* dan kedalaman *tree* adalah empat. Jumlah K_1 dan K_2 dibuat bervariasi, K_2 bernilai separuh dari K_1 dan juga K_2 bernilai sama besar dengan nilai K_1. Hasil uji coba ini seperti yang terlihat pada Gambar 5.5 untuk pendekatan pertama dan Gambar 5.6 untuk pendekatan kedua.

Terlihat dari kedua gambar, bahwa jumlah K_1 dan K_2 mempengaruhi waktu dengan mendekati kurva kuadratik. Terlihat dari kedua grafik, bahwa perbedaan yang kurang signifikan terlihat ketika K_2 bernilai separuh dari K_1 berbanding ketika K_2 bernilai sama besar dengan K_1.



Gambar 5.6 Grafik Hasil Uji Coba Pengaruh Jumlah K_1 dan K_2 Terhadap Waktu untuk Pendekatan Kedua

5.3.2.3 Uji Coba pada Situs Penilaian Daring SPOJ

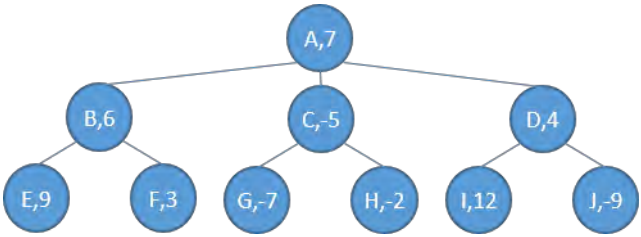
Dilakukan pengiriman kode sumber sebanyak dua puluh kali ke pada situs penilaian daring untuk mendapatkan variasi waktu dan memori yang dibutuhkan program untuk menyelesaikan permasalahan *Disjoint Subtrees*.

Dari uji coba yang telah seluruh kode sumber dengan pendekatan DP yang dikirimkan mendapat umpan balik *Accepted*. Waktu yang dibutuhkan program untuk menyelesaikan permasalahan ini adalah 0.00 detik untuk 19 percobaan dan 0.01 detik untuk percoban kelima. Memiliki rata-rata 0.0005 dengan standar deviasi 0.002236. Memori yang dibutuhkan adalah 3.4MB.

Untuk pendekatan LCRS, hasil uji coba kode sumber yang dikirimkan semuanya mendapat umpan balik *Accepted*. Dengan waktu penyelesaian 0.00 detik dan memori 3.4MB.

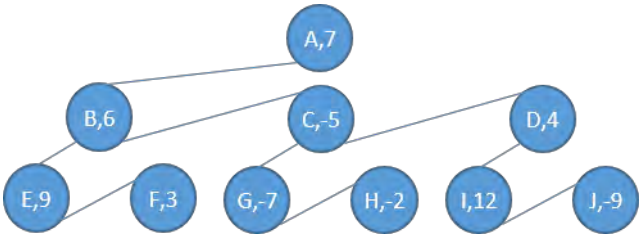
5.4 Analisis Hasil Uji Coba

Dari subbab 5.3.2 terlihat bahwa kinerja kedua pendekatan bisa dikatakan sama baiknya. Untuk lebih jelas, bisa dilihat pada simulasi bagaimana kedua pendekatan menyelesaikan permasalahan dengan *tree* seperti yang terlihat pada Gambar 5.7, dengan K_1 dan K_2 bernilai tiga.



Gambar 5.7 *Tree* Simulasi

Proses penyelesaian dengan pendekatan pertama berjalan secara DFS pada *tree* masukan, sehingga urutan pengerjaannya adalah E, F, B, G, H, C, I, J, D, dan A. Untuk proses penyelesaian dengan pendekatan LCRS dilakukan pengonversian *arbitrary tree* seperti yang terlihat pada Gambar 5.7, menjadi *LCRS tree* seperti yang terlihat pada Gambar 5.8. Proses penyelesaian berjalan dengan urutan *vertex* sebagai berikut: F, E, H, G, J, I, D, C, B, dan A.



Gambar 5.8 *LCRS Tree* Simulasi

Bila proses penggabungan nilai submasalah pada *children* pada pendekatan pertama diubah urutannya, dimulai dari child terakhir

hingga child pertama seperti yang terlihat pada Tabel 5.1, dibandingkan dengan nilai pada *vertex* B, C, dan D pada penyelesaian dengan pendekatan kedua seperti yang terlihat pada Tabel 5.2, keduanya memiliki nilai yang sama persis. Dapat disimpulkan bahwa kedua pendekatan hanya berbeda dalam proses urutan penyelesaian *vertex*.

Tabel 5.1 Nilai Submasalah pada Proses Penggabungan *Children Vertex* A

V	3				3 & 2				3, 2, & 1			
N\K	1	2	3	4	1	2	3	4	1	2	3	4
0	0	0	-7	7	0	0	14	7	0	0	14	18
1	4	-4	INF	INF	4	5	18	12	6	5	20	23
2	16	5	INF	INF	16	12	30	19	16	12	30	30
3	7	-7	INF	INF	11	14	21	21	22	14	36	32

Tabel 5.2 Nilai Submasalah pada *Vertex* D, C, dan B dengan Pendekatan Kedua

V	D				C				B			
N\K	1	2	3	4	1	2	3	4	1	2	3	4
0	0	0	-7	7	0	0	14	7	0	0	14	18
1	4	-4	INF	INF	4	5	18	12	6	5	20	23
2	16	5	INF	INF	16	12	30	19	16	12	30	30
3	7	-7	INF	INF	11	14	21	21	22	14	36	32

Kedua pendekatan memiliki kompleksitas waktu $O(NK^2)$. Didapat dari memori untuk menyimpan nilai submasalah yang memiliki dimensi $O(K)$ dengan transisi $O(NK)$untuk setiap nilai pada memori tersebut, terlihat pada implementasi fungsi *merging* pada subbab 4.3.3.2.6 pada pendekatan DP dan implementasi fungsi *processSibling* pada subbab 4.3.4.2.8.

BAB VI
KESIMPULAN

Kesimpulan yang dapat diambil berdasarkan hasil uji coba dan evaluasi terhadap implementasi algoritma pemrograman dinamis dengan menggunakan pemrograman dinamis kembali dalam proses penggabungan submasalah *children* dan dengan menggunakan pengonversian *arbitrary tree* menjadi *left child right sibling tree* adalah sebagai berikut:

1. Implementasi algoritma pemrograman dinamis dengan pendekatan pertama, menggunakan pemrograman dinamis kembali pada proses penggabungan submasalah *children* dan pendekatan kedua, mengonversi *arbitrary tree* menjadi *left child right sibling tree* dalam menyelesaikan permasalah SPOJ *Disjoint Subtrees* dengan mendapat umpan balik *Accepted*, waktu rata-rata 0.00 detik dan memori yang dibutuhkan adalah 3.4 MB dari 20 kali pengumpulan.
2. Pemrograman dinamis pada permasalahan ini mempunyai kompleksitas waktu $O(NK^2)$ dengan N adalah jumlah *vertex* dan K adalah nilai terbesar diantara K_1 dan K_2 ini sejalan dengan hasil uji coba kinerja yang didapat dimana jumlah *vertex* mempengaruhi secara linear dan nilai terbesar diantara K_1 dan K_2 mempengaruhi secara kuadratik.

DAFTAR PUSTAKA

Halim, S., & Halim, F. (2013). *Competitive Programming* (3th ed.). Singapore.

LAMPIRAN A

Tabel 8.1 Hasil Pengujian Pendekatan DP pada situs SPOJ sebanyak 20 kali

No	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	0.00	3.4
2	Accepted	0.00	3.4
3	Accepted	0.00	3.4
4	Accepted	0.00	3.4
5	Accepted	0.01	3.4
6	Accepted	0.00	3.4
7	Accepted	0.00	3.4
8	Accepted	0.00	3.4
9	Accepted	0.00	3.4
10	Accepted	0.00	3.4
11	Accepted	0.00	3.4
12	Accepted	0.00	3.4
13	Accepted	0.00	3.4
14	Accepted	0.00	3.4
15	Accepted	0.00	3.4
16	Accepted	0.00	3.4
17	Accepted	0.00	3.4
18	Accepted	0.00	3.4
19	Accepted	0.00	3.4
20	Accepted	0.00	3.4

Tabel 8.2 Hasil Pengujian Pendekatan LCRS pada situs SPOJ sebanyak 20 kali

No	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	0.00	3.4
2	Accepted	0.00	3.4
3	Accepted	0.00	3.4
4	Accepted	0.00	3.4
5	Accepted	0.00	3.4
6	Accepted	0.00	3.4
7	Accepted	0.00	3.4
8	Accepted	0.00	3.4
9	Accepted	0.00	3.4
10	Accepted	0.00	3.4
11	Accepted	0.00	3.4
12	Accepted	0.00	3.4
13	Accepted	0.00	3.4
14	Accepted	0.00	3.4
15	Accepted	0.00	3.4
16	Accepted	0.00	3.4
17	Accepted	0.00	3.4
18	Accepted	0.00	3.4
19	Accepted	0.00	3.4
20	Accepted	0.00	3.4

BIODATA PENULIS

Penulis, Fandi A. Rahadian, lahir di Jakarta pada tanggal 18 Januari 1992. Anak pertama dari pasangan Teguh Pambudi dan Herawati. Penulis menempuh pendidikan formal mulai dari TK hingga S-1 di TKi Al-Azhar Kemang Pratama (1997-1998), SDi Al-Azhar Jaka Permai (1998-2004), SMPN 109 Jakarta (2004-2007), SMAN 81 Jakarta (2007-2010), dan Jurusan Teknik Informatika Fakultas Teknologi Informasi (FTIF) Institut Teknologi Sepuluh Nopember (ITS) Surabaya (2011-2015).